



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# 计算机图形学

## 阴影与全局光照

陶钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

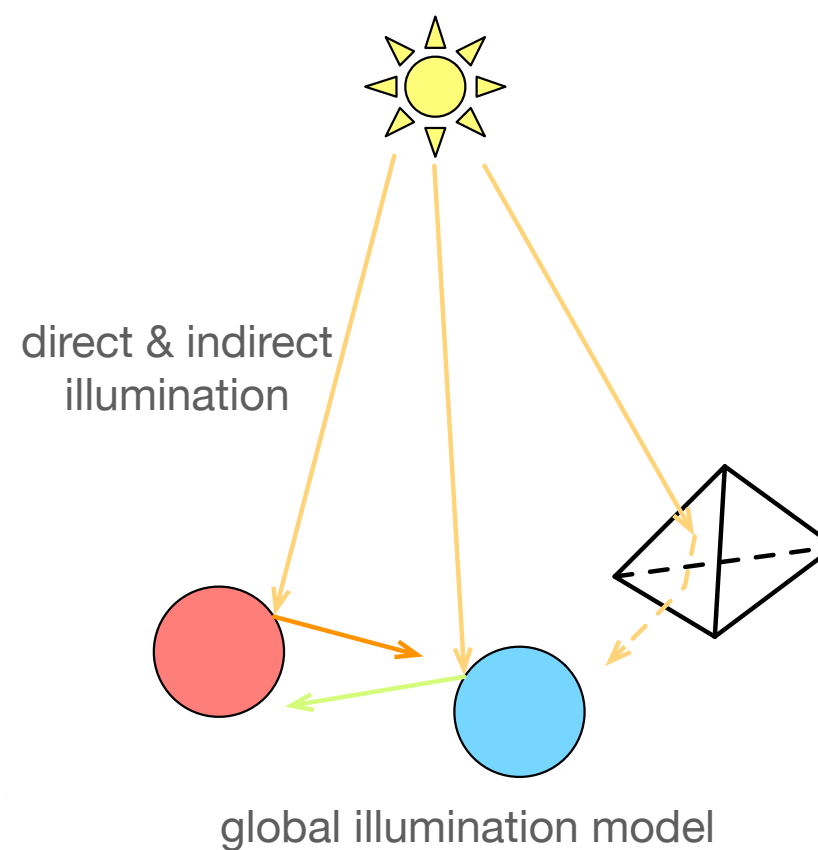
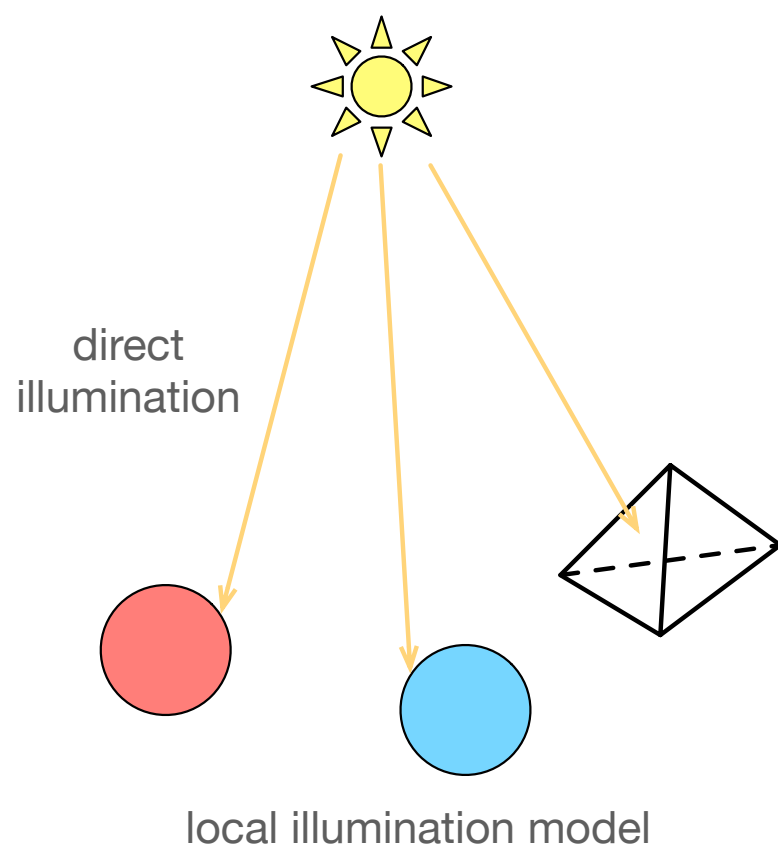
中山大学 计算机学院  
国家超级计算广州中心

- 局部光照模型下的阴影产生
- 全局光照
  - Ray tracing
  - Radiosity
- 射线与物体求交



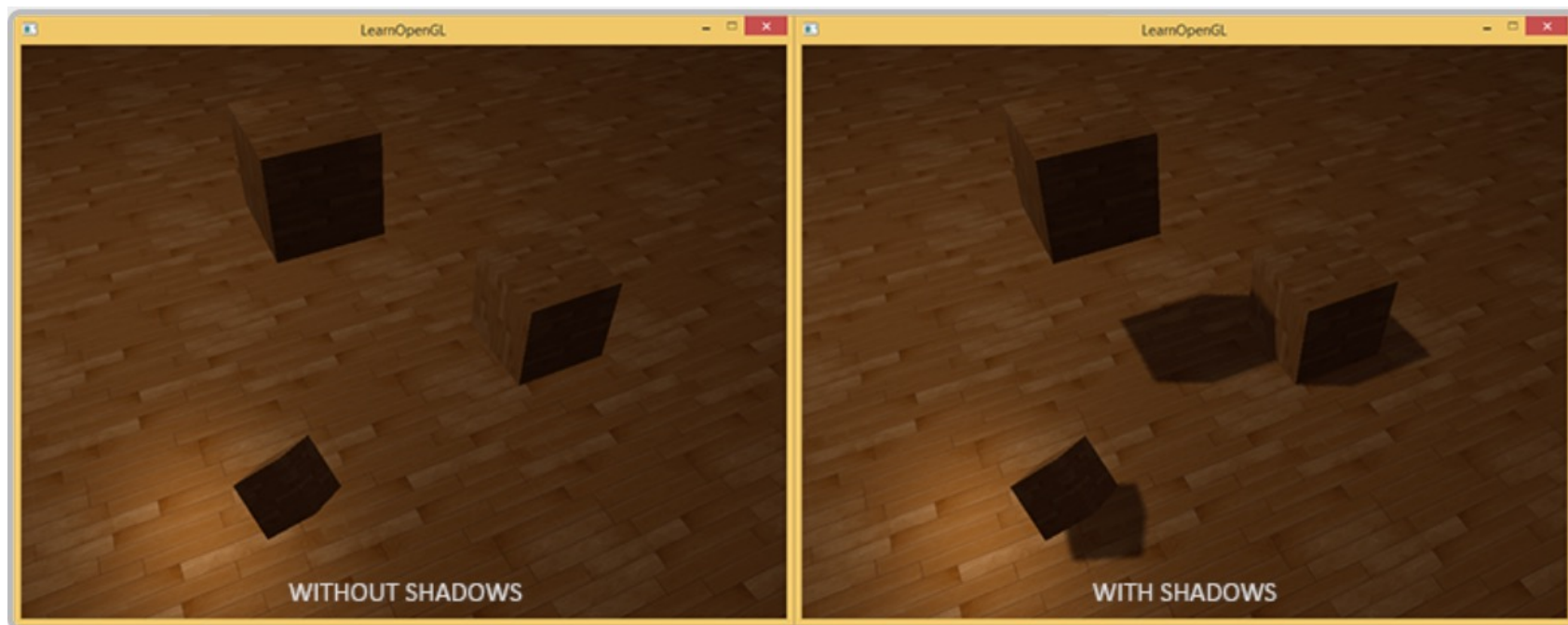
## 光照模型

- 局部光照模型：只考虑由光源直接发出的光
- 全局光照模型：考虑通过别的物体/自身对光的透射和反射



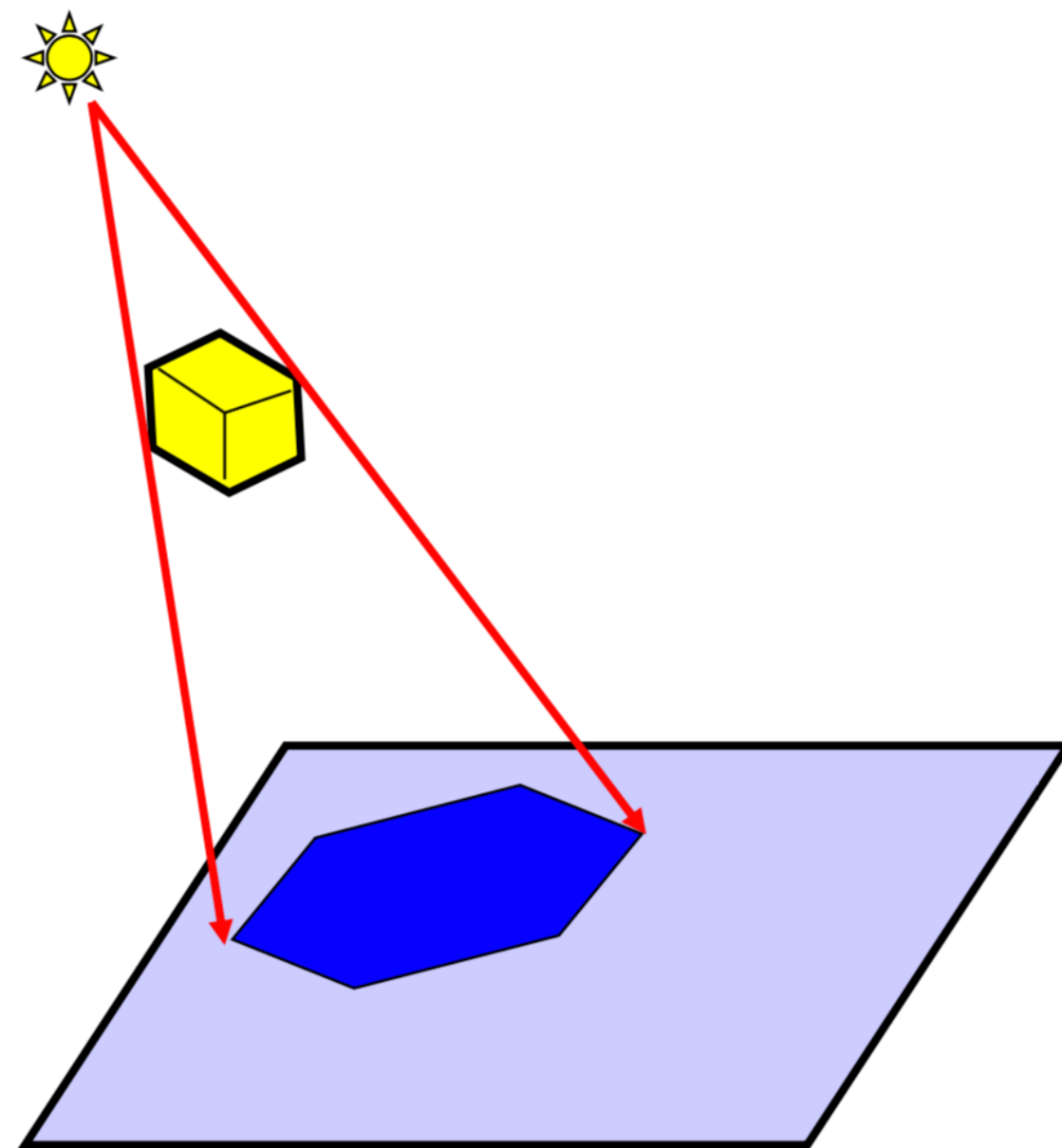
## 全局与局部照明

- 全局照明在计算过程中考虑光的实际传播过程，因此能产生阴影
- 局部照明每个点的颜色仅由该点的法向量，光照位置，观察者位置决定，因此难以产生阴影
- 是否能在局部照明的模型下产生阴影？是否可以使用OpenGL实现？
  - 能，但有一定限制



## ● 阴影产生过程

- 本质是**投影**！
  - 应该使用哪种视椎体？
- 将物体沿光照方向投影至被阴影覆盖的平面
  - 对物体上的每个顶点，将其沿光线方向投影至场景
  - 光线方向与光源类型相关
- 使用黑色绘制投影对象
- 如何高效地计算投影对象？



## 构建投影矩阵 $\text{shadowMat}[][]$

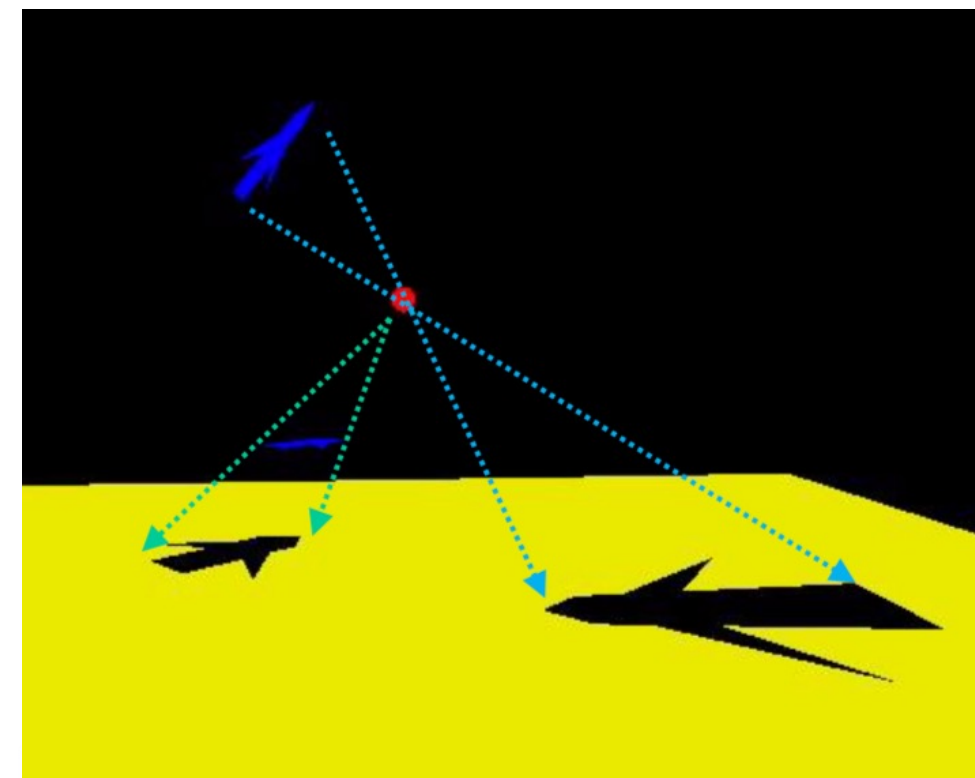
– 此处为简化版本

- 光源位于z轴上 ( $O = (0, 0, c)$ )
- 投影平面为xy平面 ( $z = 0$ )
- 物体上的一个顶点为  $A = (x, y, z)$

– 求射线  $OA$  与xy平面的交点  $P$

- $P = O + t(A - O) = (tx, ty, c + t(z - c))$
- $t = \frac{c}{c - z}$
- 写成矩阵表示形式

$$\circ P = \begin{bmatrix} u \\ v \\ 0 \\ s \end{bmatrix} = \begin{bmatrix} c & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$





## ● 使用投影变换进行绘制

- 将变换矩阵入栈
- 绘制物体
- 将变换矩阵出栈
- 将变换矩阵入栈
- 自行构建投影矩阵`shadowMat[] []`
- 将栈顶矩阵乘`shadowMat[] []`
- 绘制物体（此时物体根据`shadowMat[] []`进行变换）
- 出栈



## 使用OpenGL观察与纹理

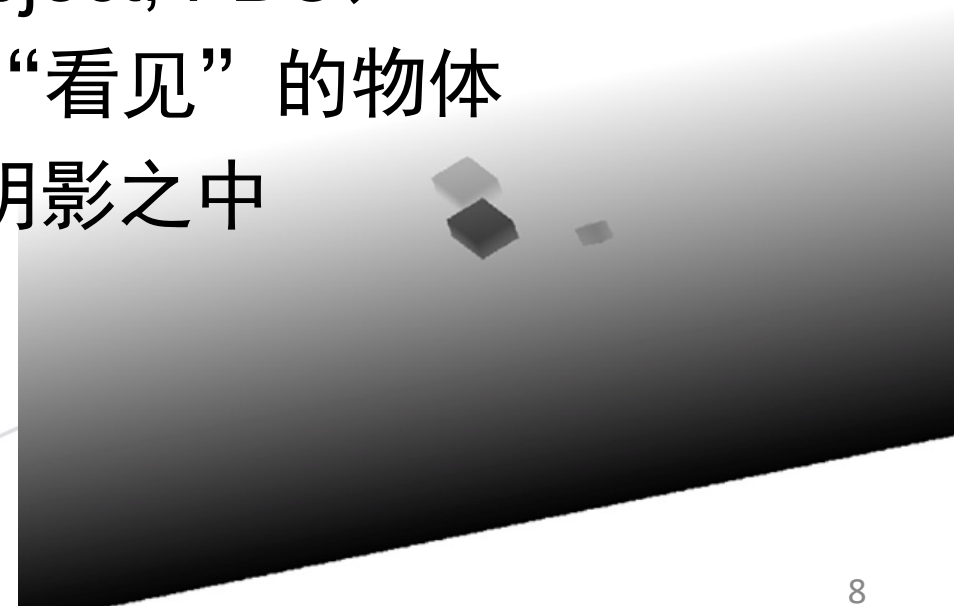
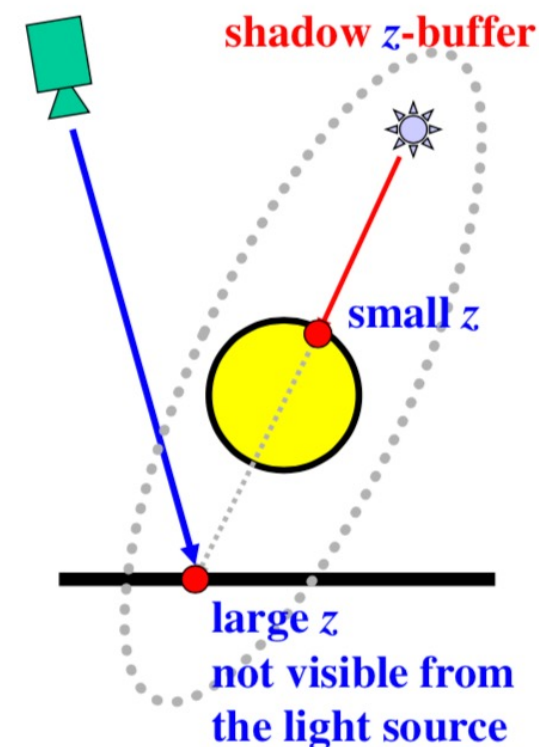
– <https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping>

### – 阴影的本质

- 从光源发射的射线在到达某个点前，先遇到其他交点
- 与camera类似！

### – 设置视角为光源视角

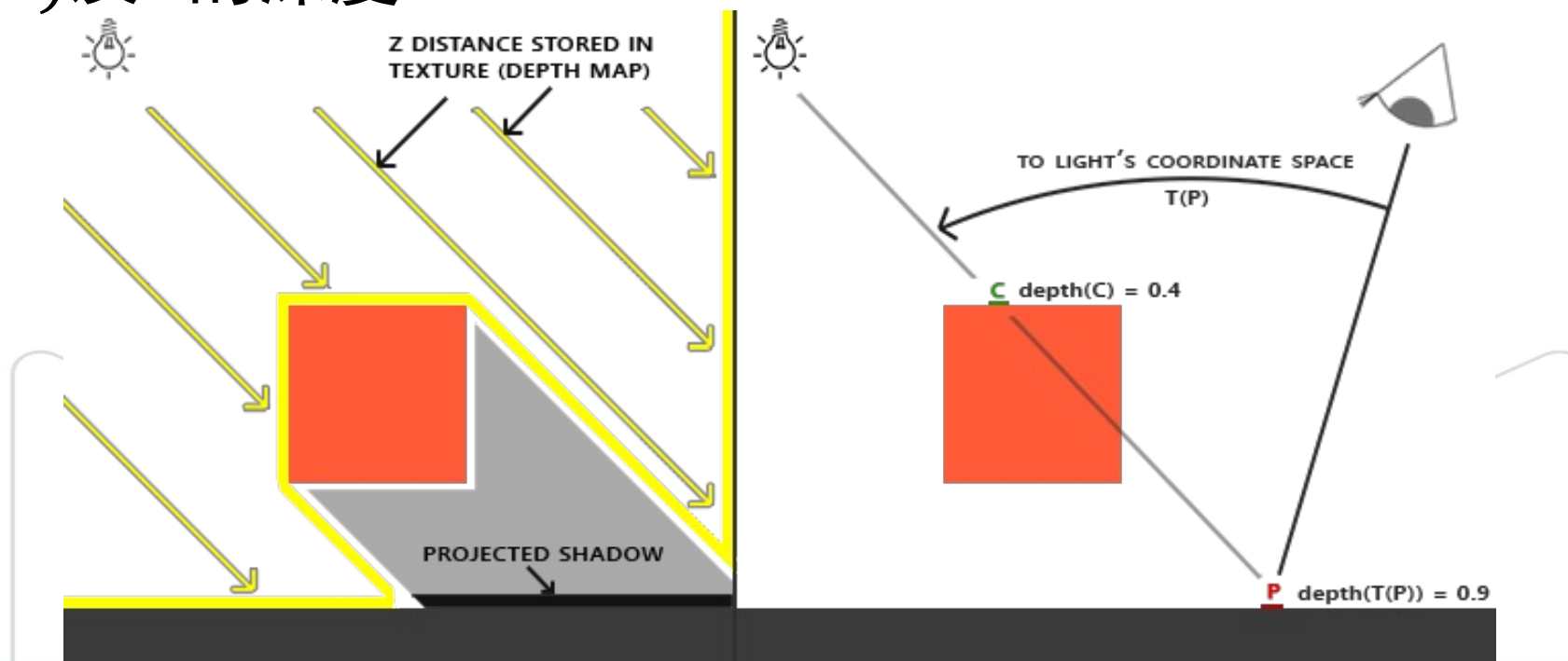
- 将深度缓冲绘制为帧缓冲对象（frame buffer object, FBO）
- 如此，则该FBO表明了以光源为视角时，所能“看见”的物体
- 利用该FBO，就能知道一个fragment是否处于阴影之中





## ● 使用OpenGL观察与纹理

- 设置视角为光源视角，以FBO为对象绘制深度缓冲
- 可根据该深度缓冲判断fragment是否处于阴影中
  - 将fragment对应的点 $P$ 变换至以光源的坐标系统中： $T(P)$
  - 获得 $T(P)$ 的深度，及其在深度缓冲中对应的像素 $C$
  - 对比 $T(P)$ 及 $C$ 的深度



## • 使用OpenGL观察与纹理

### – 创建FBO

```
unsigned int depthMapFBO;  
glGenFramebuffers(1, &depthMapFBO);
```

### – 创建纹理

```
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;  
unsigned int depthMap;  
glGenTextures(1, &depthMap);  
glBindTexture(GL_TEXTURE_2D, depthMap);  
glTexImage2D(GL_TEXTURE_2D, 0, 1, SHADOW_WIDTH, SHADOW_HEIGHT, 0,  
              GL_DEPTH_COMPONENT, GL_FLOAT, NULL);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

## ● 使用OpenGL观察与纹理

### — 将纹理attach到FBO上

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);  
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,  
                        GL_TEXTURE_2D, depthMap, 0);  
  
glDrawBuffer(GL_NONE);  
glReadBuffer(GL_NONE);  
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```



## • 使用OpenGL观察与纹理

### – 绘制depth buffer

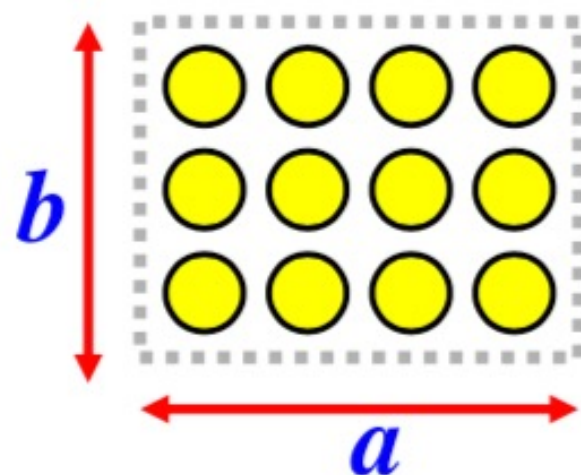
```
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);  
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFB0);  
glClear(GL_DEPTH_BUFFER_BIT);  
ConfigureShaderAndMatrices();  
RenderScene();  
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

### – 绘制实际场景

```
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);  
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
ConfigureShaderAndMatrices();  
glBindTexture(GL_TEXTURE_2D, depthMap);  
RenderScene();
```

- 产生具有真实感的阴影

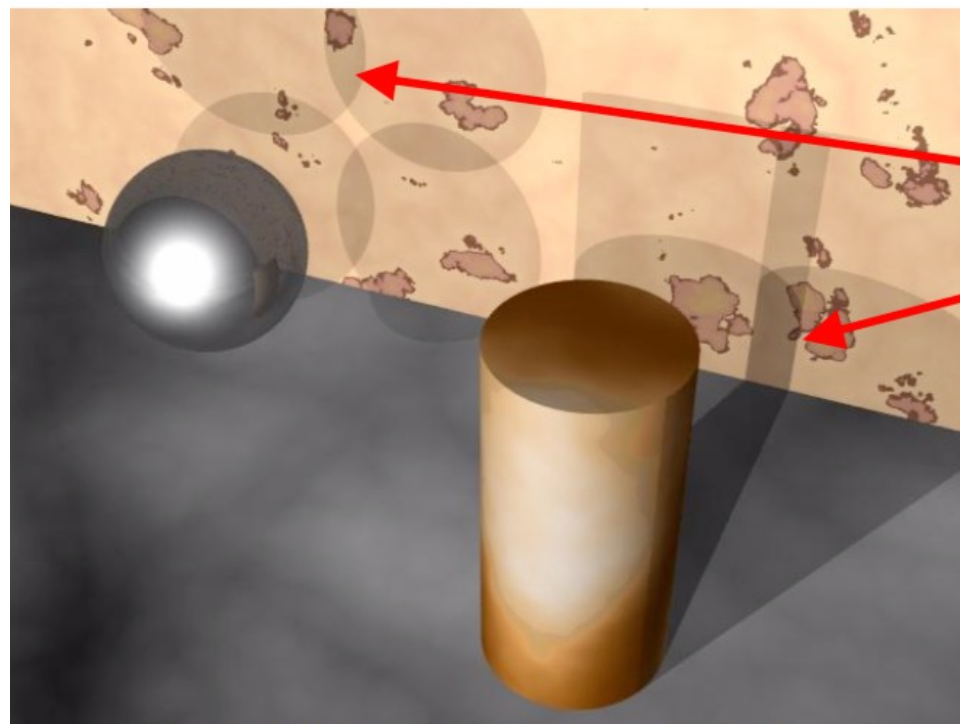
- 光线追踪产生的阴影具有清晰的边界 (hard shadow)
  - 这节课介绍的投影方法本质上为光线追踪
- Radiosity与photo-mapping产生soft shadow
- 如何使用现有算法产生soft shadow?
  - 使用多个光源模拟区域光源!



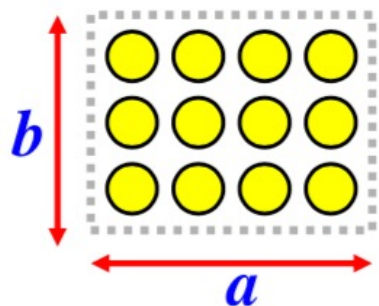
size:  $a \times b$   
array:  $4 \times 3$



- 产生具有真实感的阴影



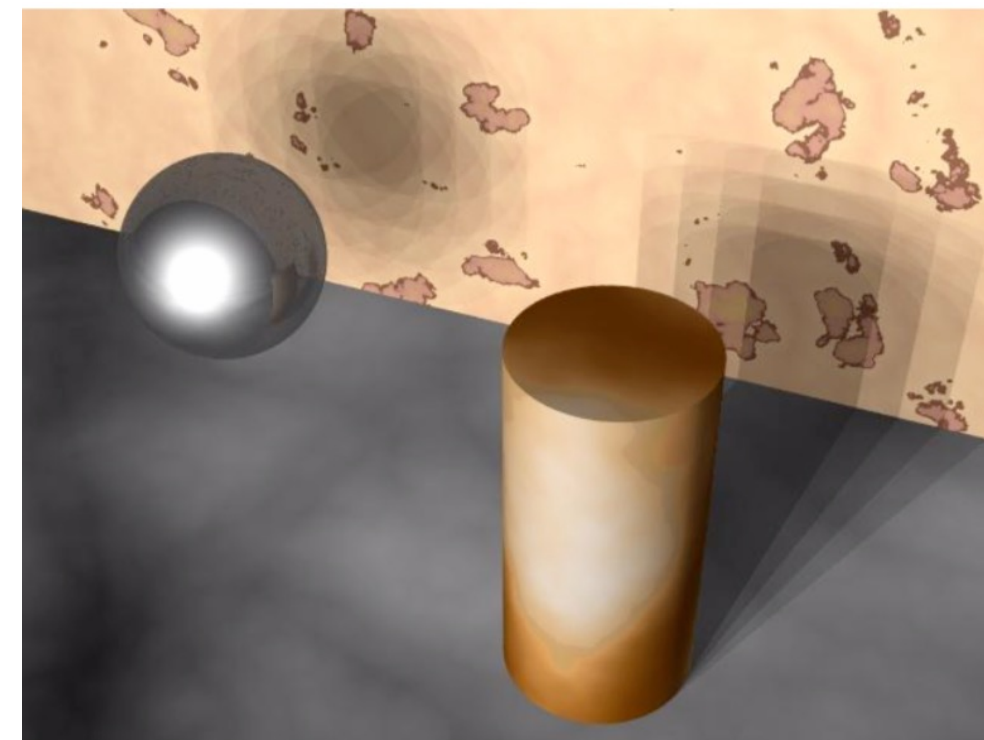
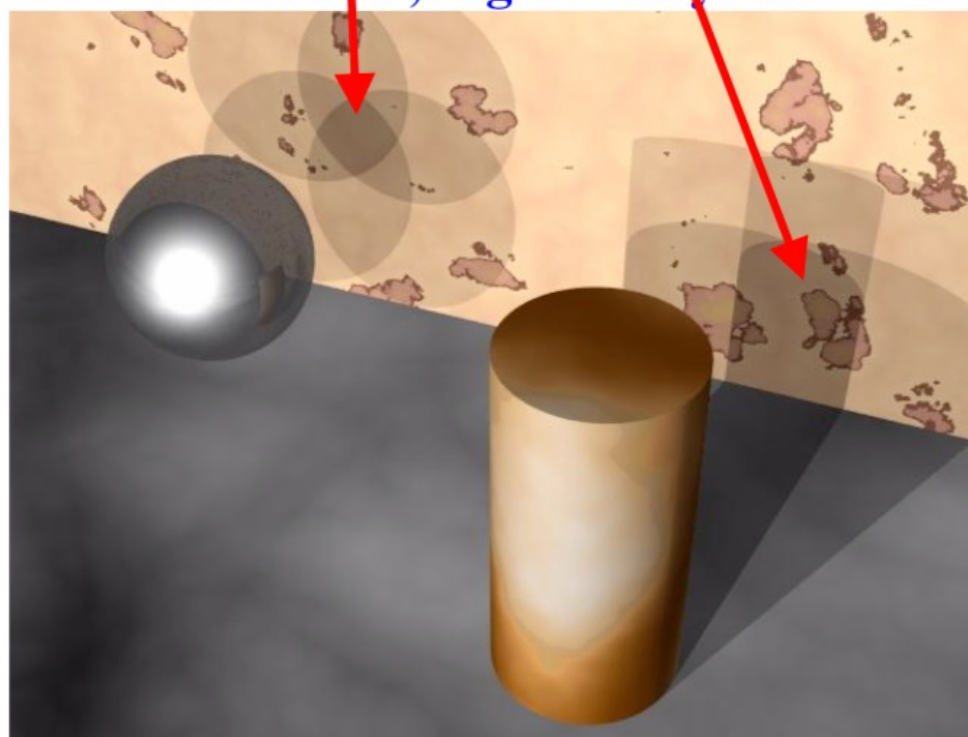
Size: 5×5, Light Array: 2×2



size:  $a \times b$   
array:  $4 \times 3$

check the overlapping shadows

Size: 3×3, Light Array: 2×2

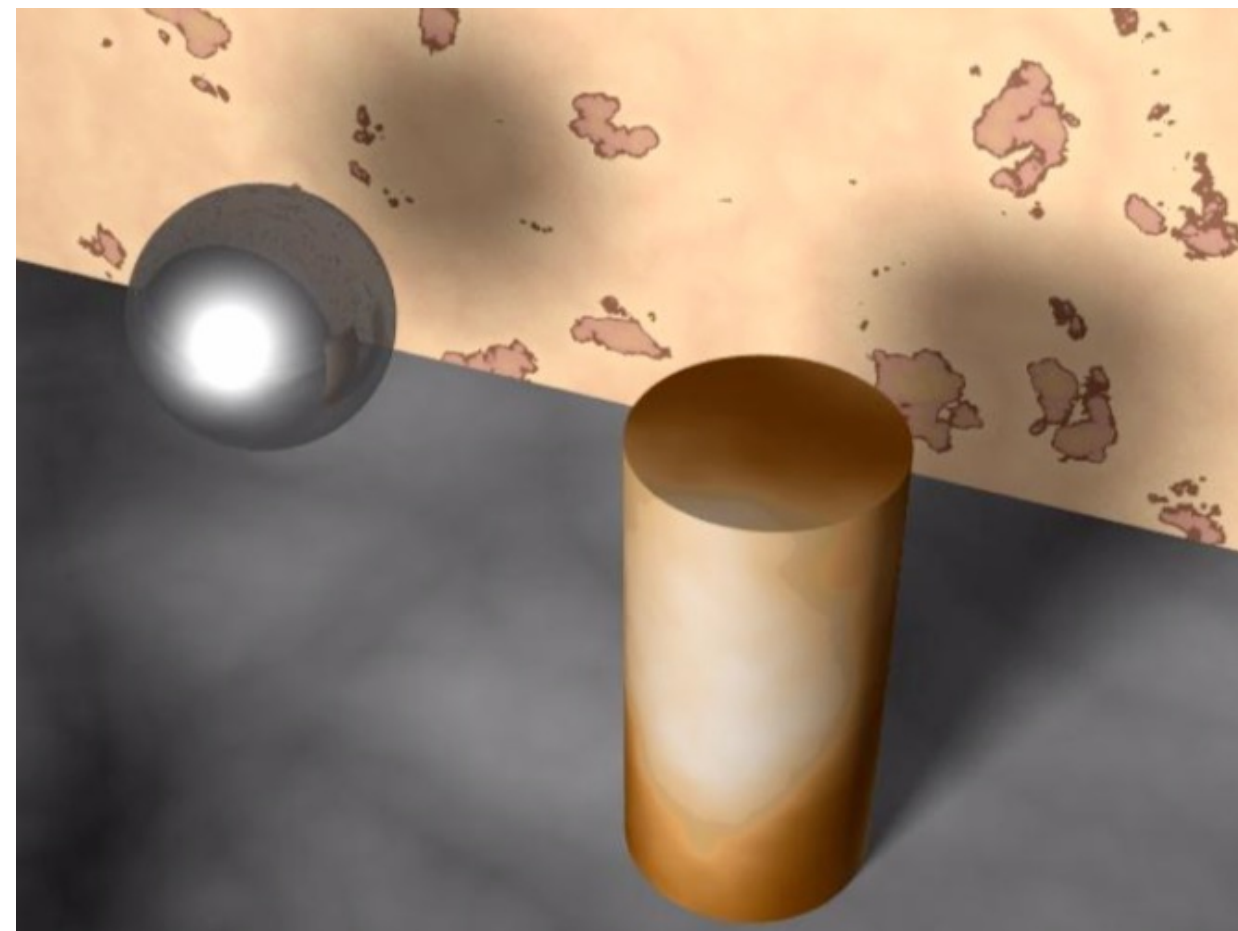
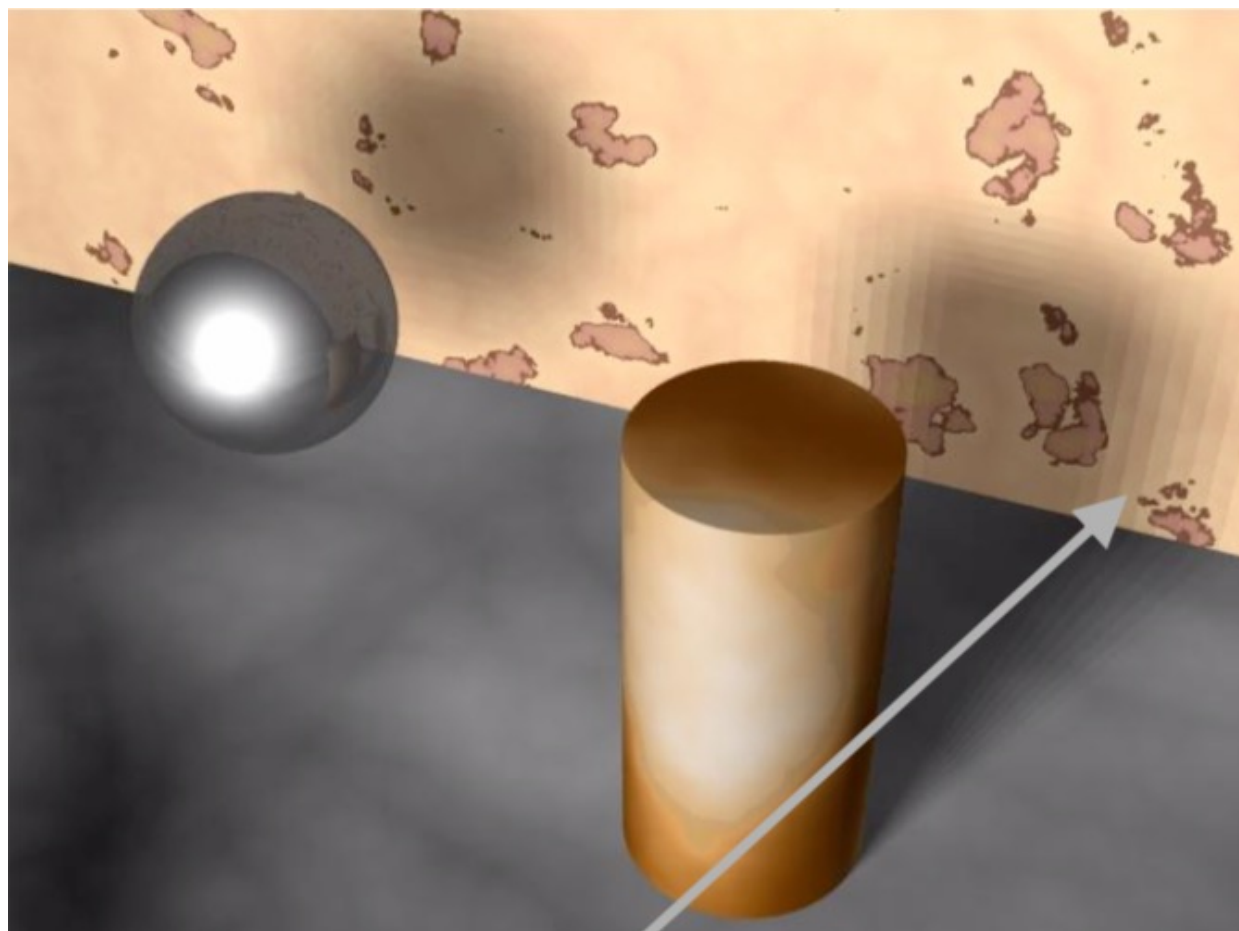


Size 3×3, Light Array: 5×5



- 产生具有真实感的阴影

- 增加光源的数量可以提高阴影的分辨率，从而产生更真实的效果
  - 但整齐排列的光源产生的阴影仍然具有颗粒感
- 使用jitter给光源排列加入微小抖动让阴影过渡更自然



- 局部光照模型下的阴影产生
- 全局光照
  - Ray tracing
  - Radiosity
- 射线与物体求交



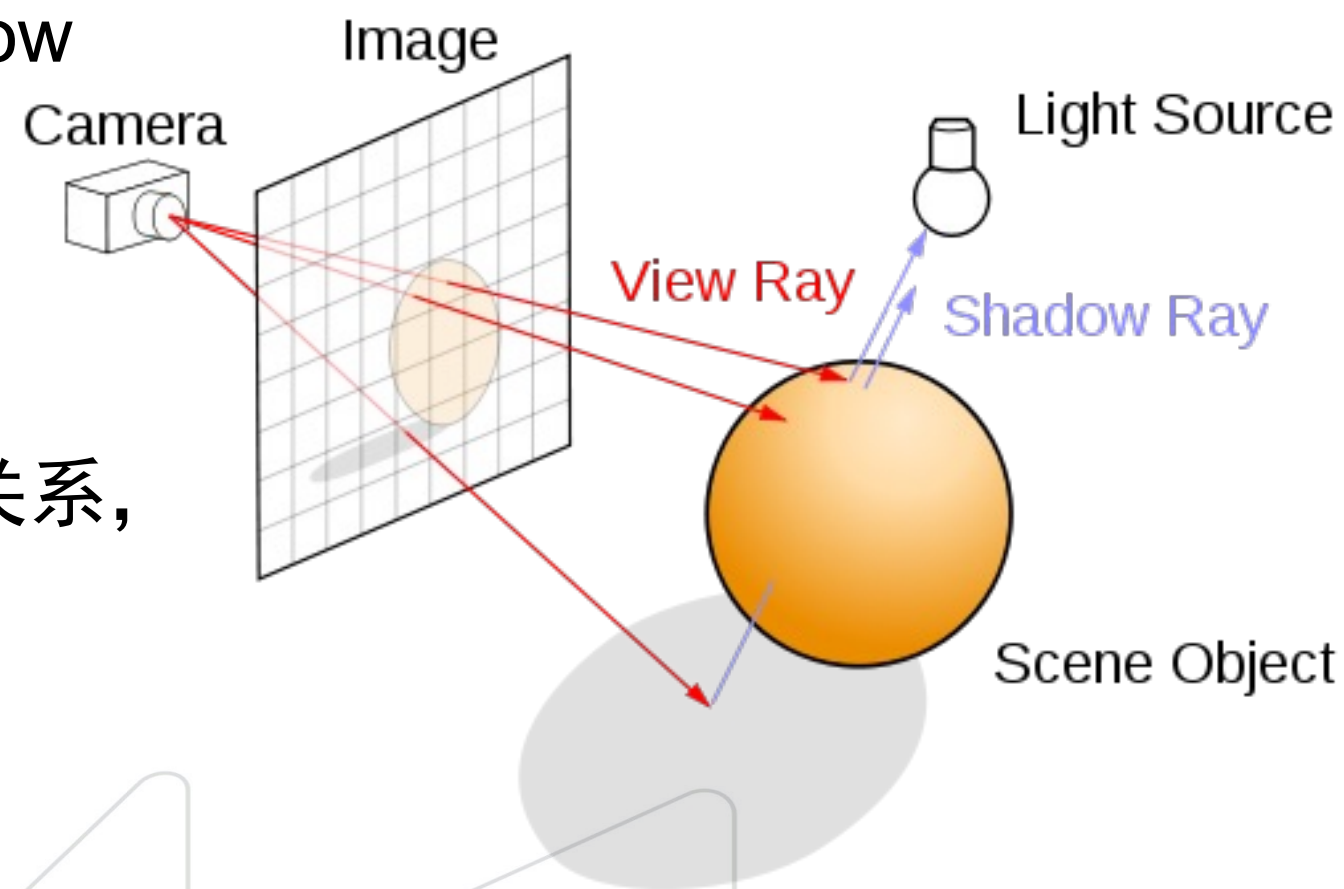
## 全局光照模型

– 局部光照模型无法产生具有真实感的光照细节

- 间接光照, color bleeding, soft shadow

– 产生全局光照的基本思路

- Ray tracing: 从视角出发追踪光线
- Photon mapping: 从光源出发追踪
- Radiosity: 计算表面上小块之间的关系, 求解每个小块的亮度

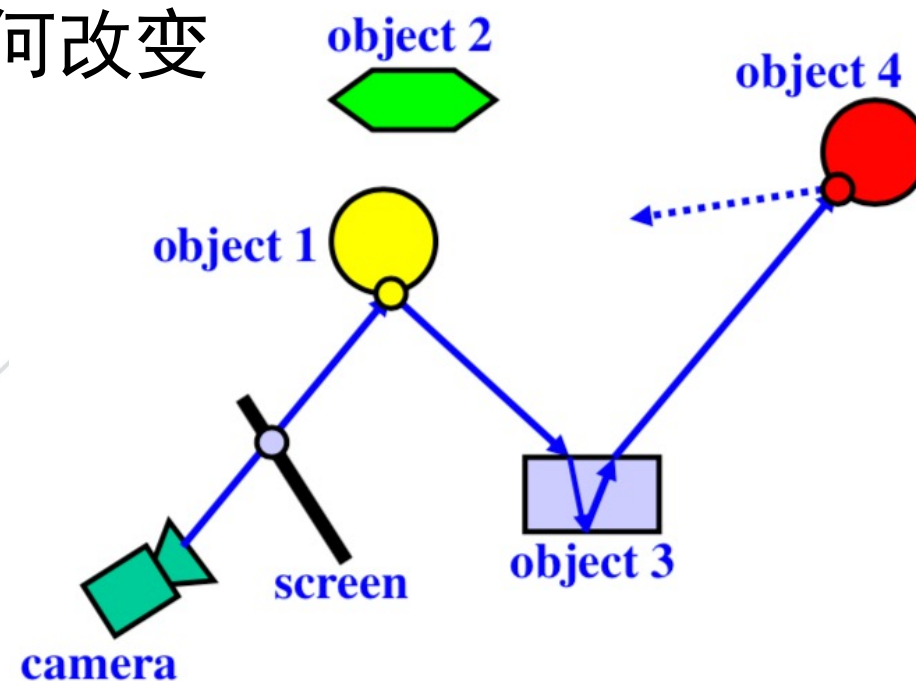


- 局部光照与全局光照简要回顾
- 全局光照
  - Ray tracing
  - Radiosity
- 射线与物体求交



## ◉ Ray tracing

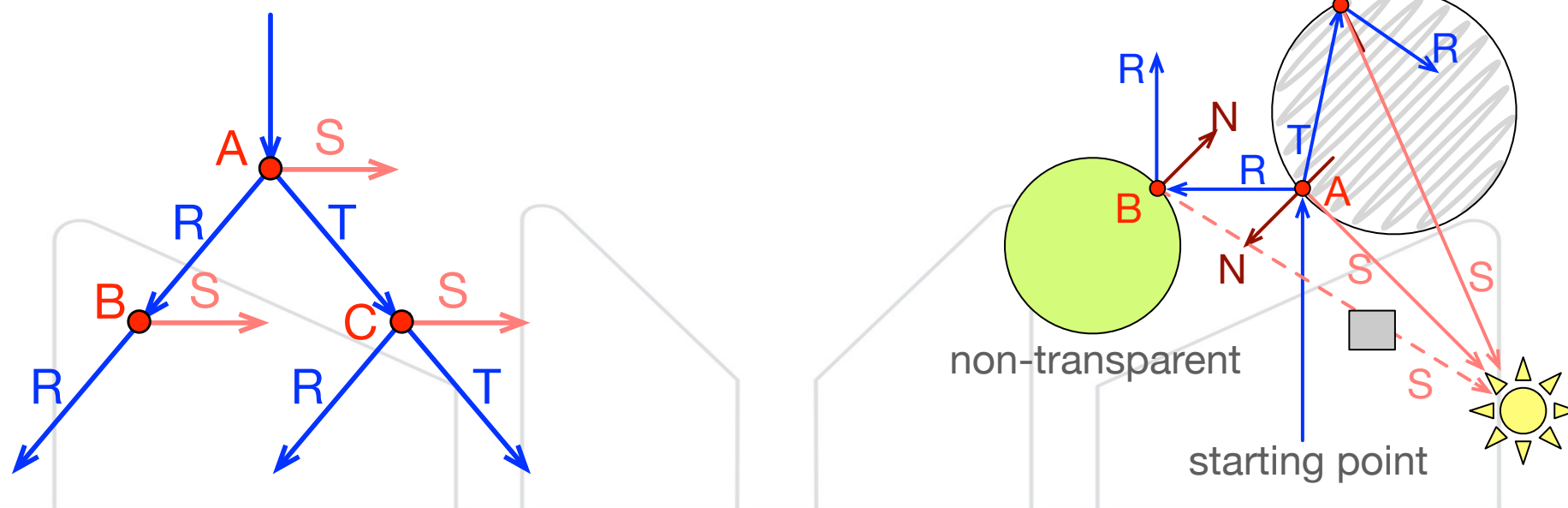
- 光线追踪又称为递归光线追踪（recursive ray tracing）
- 从观察视角出发递归地追踪光线来源
  - 在碰到物体时会产生反射与折射
  - 当光线打在某个物体上时，光线会带上该物体的颜色
  - 如果所追踪的光线没有碰到任何物体，则该视线上没有任何物体，因此该点的像素颜色（底色）也不会发生任何改变





## Ray tracing

- 初始化时产生与图像像素相同数目的光线数目进行追踪
  - 当光线打在物体上时，产生三道光线进行进一步追踪
    - 反射光R：在物体表面照射点反射
    - 透射光T：在物体表面照射点发生折射，穿过物体内部
    - 阴影射线S：从物体表面照射点指向光源，如果S被其他物体遮挡，则该点无法从光源接收光照





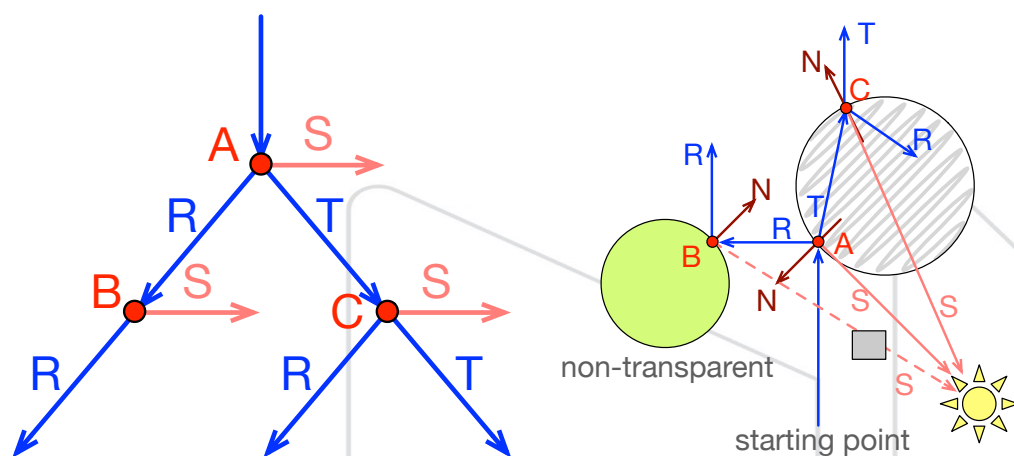
## Ray tracing

– 在Phong反射模型中，物体最终呈现出的颜色为

$$\bullet I = k_a L_a + \sum_i \left( k_d L_d (\mathbf{n} \cdot \mathbf{l}_i) + k_s L_s \cdot \max((\mathbf{n} \cdot \mathbf{h}_i)^\alpha, 0) \right)$$

– 而在光线追踪算法中，由于引入反射光，折射光与阴影射线，对光照的反射模型也许要随之变化


$$\bullet I = k_a L_a + \sum_i S_i \left( k_d L_d (\mathbf{n} \cdot \mathbf{l}_i) + k_s L_s \cdot \max((\mathbf{n} \cdot \mathbf{h}_i)^\alpha, 0) \right) + k_r I_r + k_t I_t$$



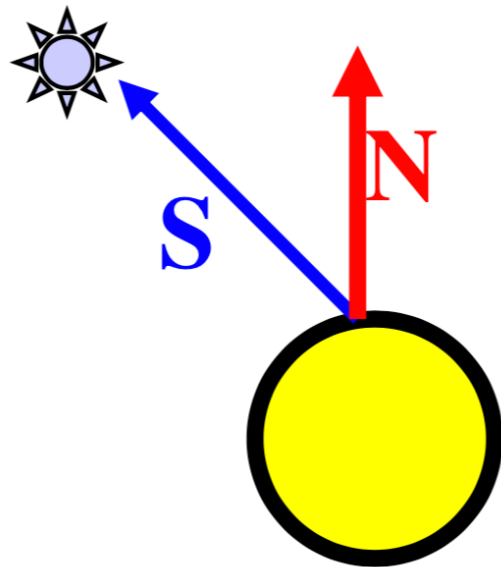
## Ray tracing

```
for each scan line do
  for each pixel on this scan line {
    ray := the ray from camera through the pixel;
    pixel-color = RayTrace(ray, 1);
  }

function RayTrace(ray, depth)
{
  if ray hits an object {
    compute intersection point P and its normal vector N;
    RayTrace := Shading(object-hit, ray, P, N, depth);
  } else {
    RayTrace := background color;
  }
}
```



## Ray tracing

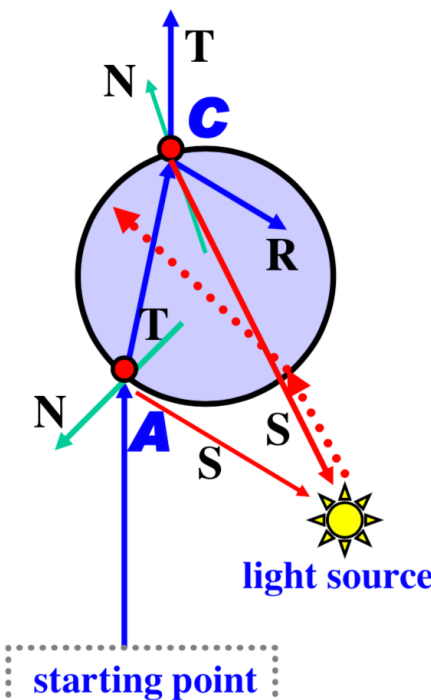
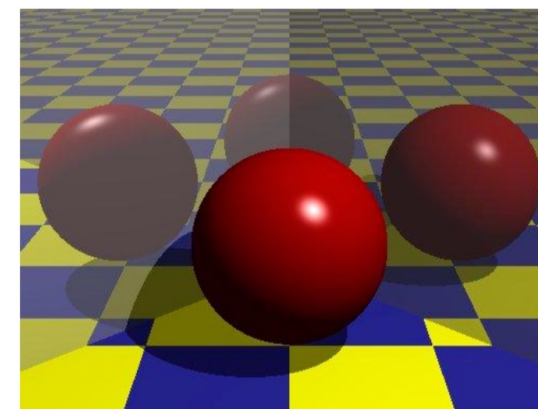
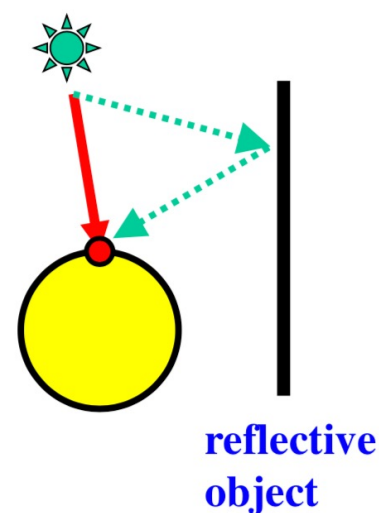
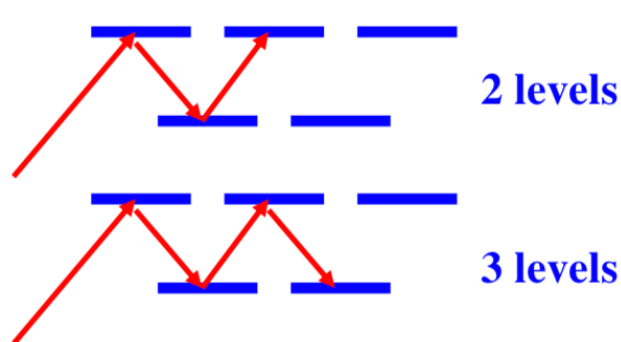


```
function Shading(Obj, iRay, P, N, Depth) {  
    color := ambient color;  
    for each light source {  
        S := shadow ray to that light source  
        if (S·N > 0) // a front point!  
            determine shadow, diffuse, specular terms of color;  
    }  
    if Depth < MaxDepth {  
        if Obj is reflective {  
            R := reflection ray direction;  
            NewColor := RayTrace(R, Depth+1);  
            scale NewColor by reflection coefficient kr and add to color;  
        }  
        if Obj is transparent {  
            T := transmitting ray direction;  
            if not total internal reflection {  
                NewColor := RayTrace(T, Depth+1);  
                scale NewColor by transmission coefficient kt and add to color;  
            }  
        }  
    }  
    Shading := color;  
}
```

## Ray tracing

### – 存在问题

- 阴影射线在发生折射时计算可能不正确
  - 右图中光源并非通过阴影射线S照射点C，而是折射后的路径
  - 因此，传统光线追踪难以产生真实的焦散（caustic）效果
- 阴影射线没考虑间接光照
  - 只考虑照射点与光源之间的关系
  - 图中缺少了镜子反射光所产生的阴影
- 追踪深度限制
  - 增加深度带来的开销可能呈几何增长



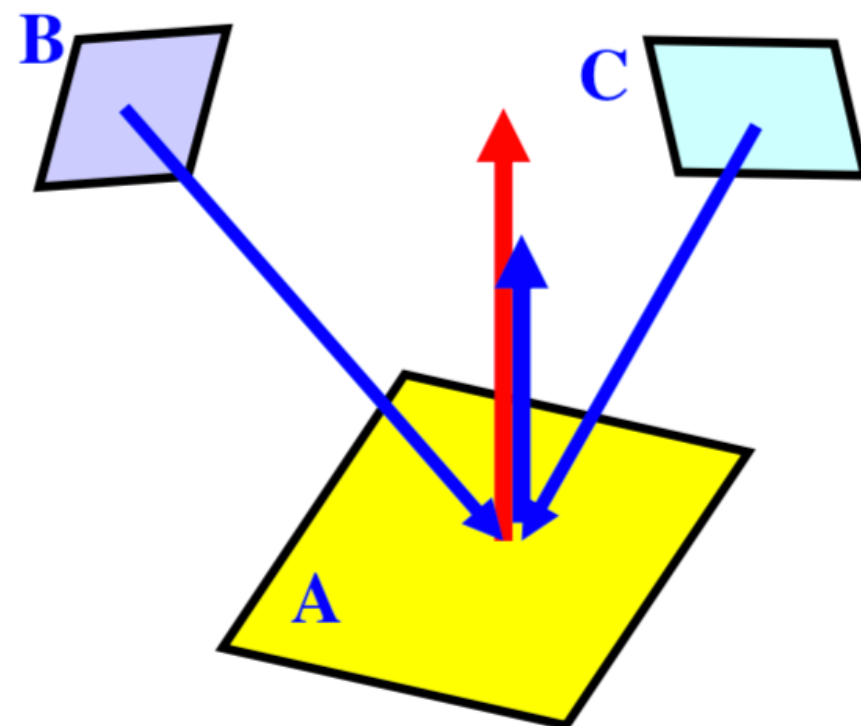
- 局部光照与全局光照简要回顾
- 全局光照
  - Ray tracing
  - Radiosity
- 射线与物体求交





## • Radiosity

- 传统ray tracing将所有追踪产生的间接光照外的全局照明都表示为无方向的环境光
- Radiosity（辐射度算法）则是以能量传播为基础构建的模型
  - 考虑所有物体表面上发射，反射，吸收的能量，及其相互之间的关系
  - 离开物体表面的能量称为radiosity
    - 自身发射能量及反射或透射其他物体表面传递过来的能量之和
    - 如图所示，A自身发射的红光及反射B和C的蓝光能量之和为A的radiosity





## ◉ Radiosity: 能量方程

$$- B_i(\Delta A_i) = E_i(\Delta A_i) + \rho_i \int_j B_j(\Delta A_j) F_{\Delta A_j \rightarrow \Delta A_i}$$

- $B_i$ : patch  $i$  当前的radiosity
- $E_i$ : patch  $i$  所发射的能量
- $\Delta A_i$ : patch  $i$  上的一个极小的区域
- $\rho_i$ : patch  $i$  反射能量的比率
- $F_{\Delta A_j \rightarrow \Delta A_i}$ : 离开  $\Delta A_j$  的能量中到达  $\Delta A_i$  的比率, 称为form factor

### – 将场景离散化为n个平面小块

- 假设radiosity在每个小块上都是均匀分布 (小块之间radiosity不同)
- 能量方程可重写为  $B_i A_i = E_i A_i + \rho_i \sum_{j=1}^n B_j A_j F_{ji}$ 
  - 此处,  $F_{ji}$  表示从小块  $j$  到小块  $i$  的form factor

## • Radiosity: 能量方程

– 由于能量在两个小块间的交换取决于小块之间的相对位置，因此  $F_{ij}A_i = F_{ji}A_j$ （即从小块*i*到*j*传输的能量等于从小块*j*到*i*的能量）

- $B_i A_i = E_i A_i + \rho_i \sum_{j=1}^n B_j A_j F_{ji} = E_i A_i + \rho_i \sum_{j=1}^n B_j A_i F_{ij}$

- 等式两边同时除以 $A_i$ ，可得

$$B_i = E_i + \rho_i \sum_{j=1}^n B_j F_{ij}$$

- 移项将 $B$ 与 $E$ 分别置于等式两边，可得

$$B_i - \rho_i \sum_{j=1}^n B_j F_{ij} = E_i$$

- 合并两个 $B_i$ 项，可得

$$(1 - \rho_i F_{ii}) B_i - \rho_i \sum_{j \neq i} B_j F_{ij} = E_i$$

## • Radiosity: 能量方程

– 将等式  $(1 - \rho_i F_{ii})B_i - \rho_i \sum_{j \neq i} B_j F_{ij} = E_i$  写成矩阵形式, 则有

$$\cdot \begin{bmatrix} 1 - \rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1 - \rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1 - \rho_n F_{nn} \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

– 每个小块发射的能量  $E_i$  及其反射能量的比率  $\rho_i$  均为给定参数

– Form factors  $F_{ij}$  通过对小块之间的几何位置计算得到

- 对于convex表面, 由于离开小块  $i$  的能量不会打在  $i$  本身, 因此  $F_{ii} = 0$

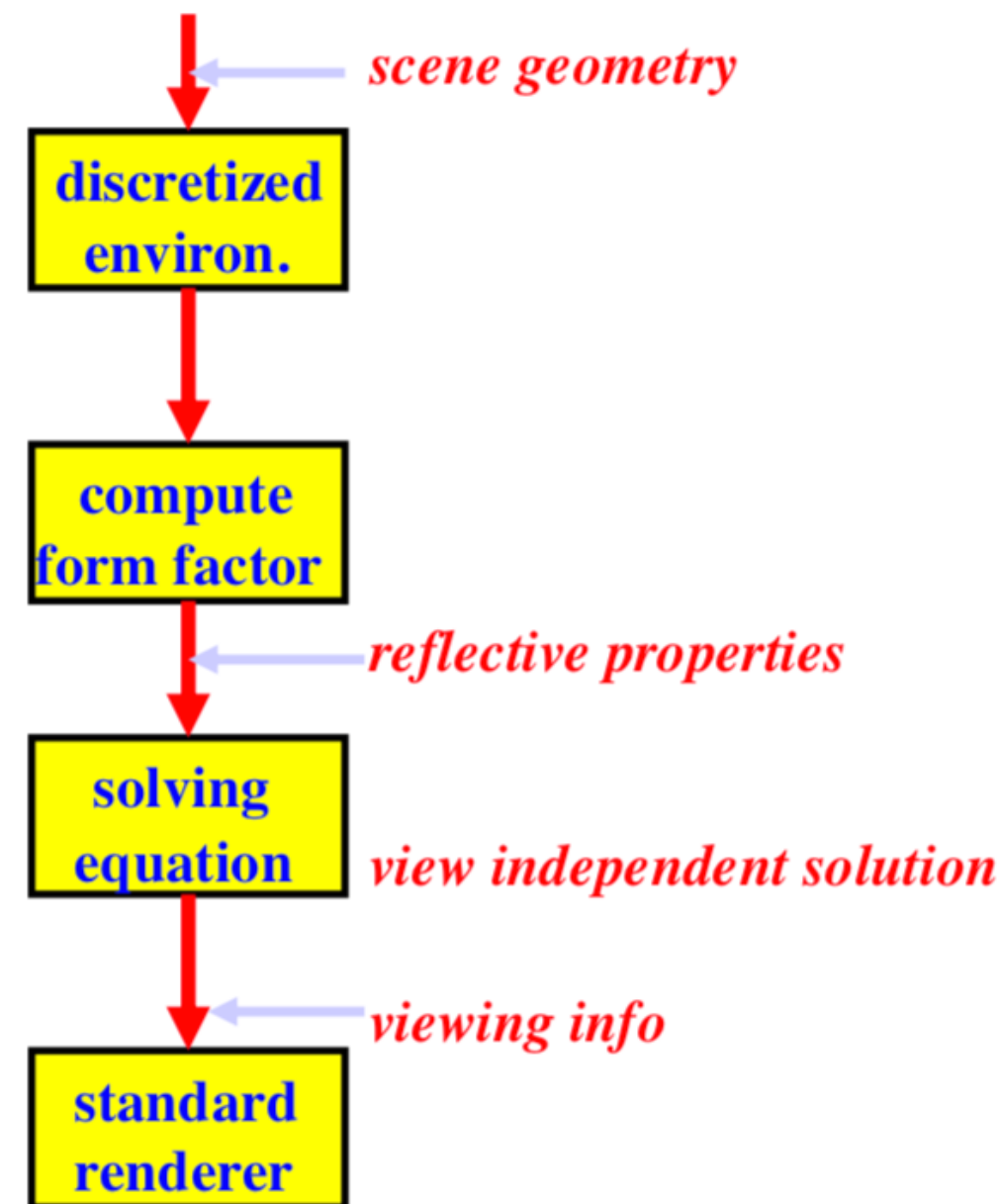
– 场景中每个小块的能量 (颜色)  $B_i$  可由求解以上线性方程组得到

- 注意, 此过程中没有引入任何与观察视角相关的信息, 因此, radiosity 是一个视角无关 (view independent) 的算法

## • Radiosity: 计算步骤

- 1. 将场景离散化为平面小块
- 2. 计算小块间能量传输的比例, 即 form factors  $F_{ij}$
- 3. 设置小块属性: 发射能量  $E_i$  及反射能量比例  $\rho_i$
- 4. 求解线性方程组, 得到每个小块的呈现出来的能量  $B_i$
- 5. 使用  $B_i$  渲染场景
  - 实际渲染中, 常加入环境光

$$B_i^{display} = B_i + \rho_i B_{ambient}$$



## • Radiosity: 求解方程组 (Gauss-Seidel Relaxation)

– 由于radiosity方程组对应的矩阵很大，通常使用迭代方法求解

- 此处我们介绍Gauss-Seidel relaxation方法
- 假设方程组 $Ax = b$ 中第 $i$ 个等式为

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{ii}x_i + \cdots + a_{in}x_n = b_i$$

- 将除 $a_{ii}x_i$ 外的所有项移至等式右边

$$a_{ii}x_i = b_i - \sum_{j=1, j \neq i}^n a_{ij}x_j$$

- 等式两边同时除以 $a_{ii}$

$$x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1, j \neq i}^n a_{ij}x_j \right)$$

## • Radiosity: 求解方程组 (Gauss-Seidel Relaxation)

– 于是，我们有了通过迭代更新 $x_i$ 的方程

$$x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1, j \neq i}^n a_{ij} x_j \right)$$

– 从一个初始状态 $X^0 = [x_1^0, x_2^0, \dots, x_n^0]$ 出发，不断迭代更新 $X$

– 在第 $i$ 次迭代，通过 $X^{i-1}$ 计算 $X^i$

– 不断进行迭代，直到系统收敛 ( $X$ 不再变化)

– 对于radiosity而言，其能量初始状态为每个小块自身发射的能量

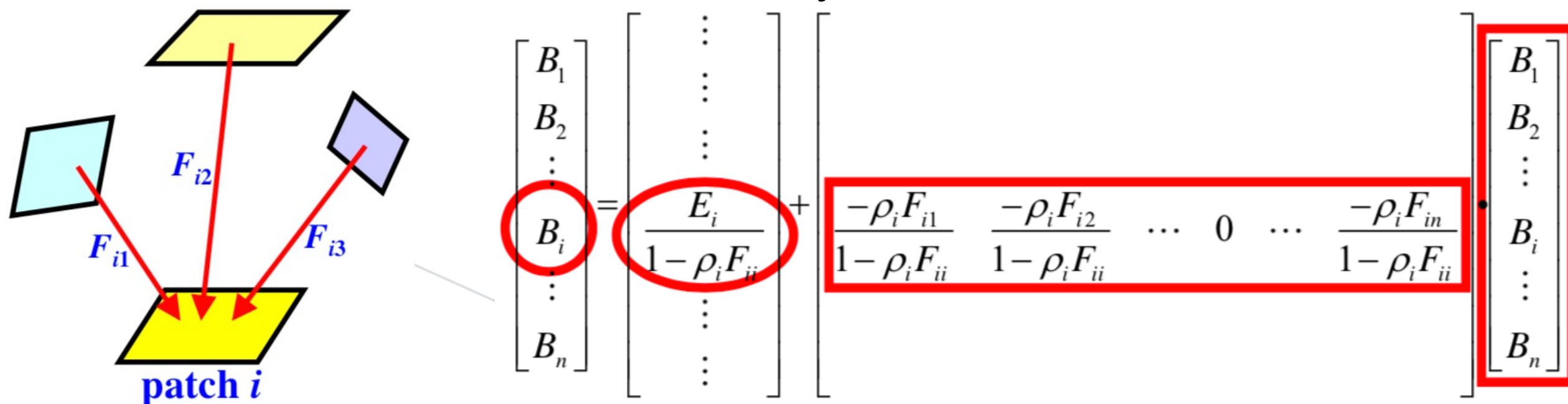
$$B^0 = [E_1, E_2, \dots, E_n]$$



## • Radiosity: 求解方程组 (Gauss-Seidel Relaxation)

– 使用Gauss-Seidel方法求解方程组在radiosity中的直观解释为

- 在每次迭代中，小块*i*呈现出的能量为其自身发射的能量，及从其他所有小块中得到的能量之和
- 在不断的能量交换迭代中，系统最终收敛至一个稳定状态：每个小块接收和发射的能量处于平衡状态，因此呈现的能量保持不变
- 使用Gauss-Seidel方法求radiosity的过程为**收集能量**的过程



$$\begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_i \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \\ \frac{E_i}{1 - \rho_i F_{ii}} \\ \vdots \\ \vdots \end{bmatrix} + \begin{bmatrix} \frac{-\rho_i F_{i1}}{1 - \rho_i F_{ii}} & \frac{-\rho_i F_{i2}}{1 - \rho_i F_{ii}} & \dots & 0 & \dots & \frac{-\rho_i F_{in}}{1 - \rho_i F_{ii}} \end{bmatrix} \cdot \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_i \\ \vdots \\ B_n \end{bmatrix}$$

## • Radiosity: 求解方程组 (Gauss-Seidel Relaxation)

– 实际计算中, 为了加快收敛速度, 常使用改进的Gauss-Seidel方法 (Southwell relaxation)

- 每次迭代中选择residual (即estimation error) 最大的等式进行更新
- 注意, 等式 $i$ 为  $\sum_{j=1}^n a_{ij}x_j = b_i$ , 即  $b_i - \sum_{j=1}^n a_{ij}x_j = 0$ , 其residual为

$$e_i = b_i - \sum_{j=1}^n a_{ij}x_j$$

- 迭代方程可写为

$$x_i^* = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1, j \neq i}^n a_{ij}x_j \right) = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^n a_{ij}x_j \right) + x_i = \frac{e_i}{a_{ii}} + x_i$$

## • Radiosity: 求解方程组 (Gauss-Seidel Relaxation)

– 迭代后  $x_i^* = \frac{e_i}{a_{ii}} + x_i$ ,  $e_i^* = 0$ , 同时可更新其他等式的residual

$$\begin{aligned} e_k^* &= b_k - \sum_{j=1, j \neq i}^n a_{kj} x_j - a_{ki} x_i^* \\ &= b_k - \sum_{j=1, j \neq i}^n a_{kj} x_j - a_{ki} \left( x_i + \frac{e_i}{a_{ii}} \right) \\ &= \left( b_k - \sum_{j=1}^n a_{kj} x_j \right) - \frac{a_{ki}}{a_{ii}} e_i \\ &= e_k - \frac{a_{ki}}{a_{ii}} e_i \end{aligned}$$

- Radiosity: 求解方程组 (Gauss-Seidel Relaxation)
  - 使用Southwell relaxation求解方程组伪码

```
for ( $i = 1$ ;  $i \leq n$ ;  $i++$ ) { // initialization
     $B_i = 0$ ;  $e_i = E_i$ ;
}
while (not converge) {
    find the  $i$  such that  $|e_i|$  is the largest;
     $B_i = B_i + e_i / (1 - \rho_i F_{ii})$ ; // update  $B_i$ 
     $temp = e_i$ ; // save  $e_i$  for later use
    for ( $k = 1$ ;  $k \leq n$ ;  $k++$ ) // update residuals
         $e_k = e_k + \rho_k F_{ki} / (1 - \rho_i F_{ii}) * temp$ ;
}
```

## • Radiosity: 求解方程组 (Gauss-Seidel Relaxation)

– 使用Southwell relaxation求解方程组的在radiosity中的直观解释

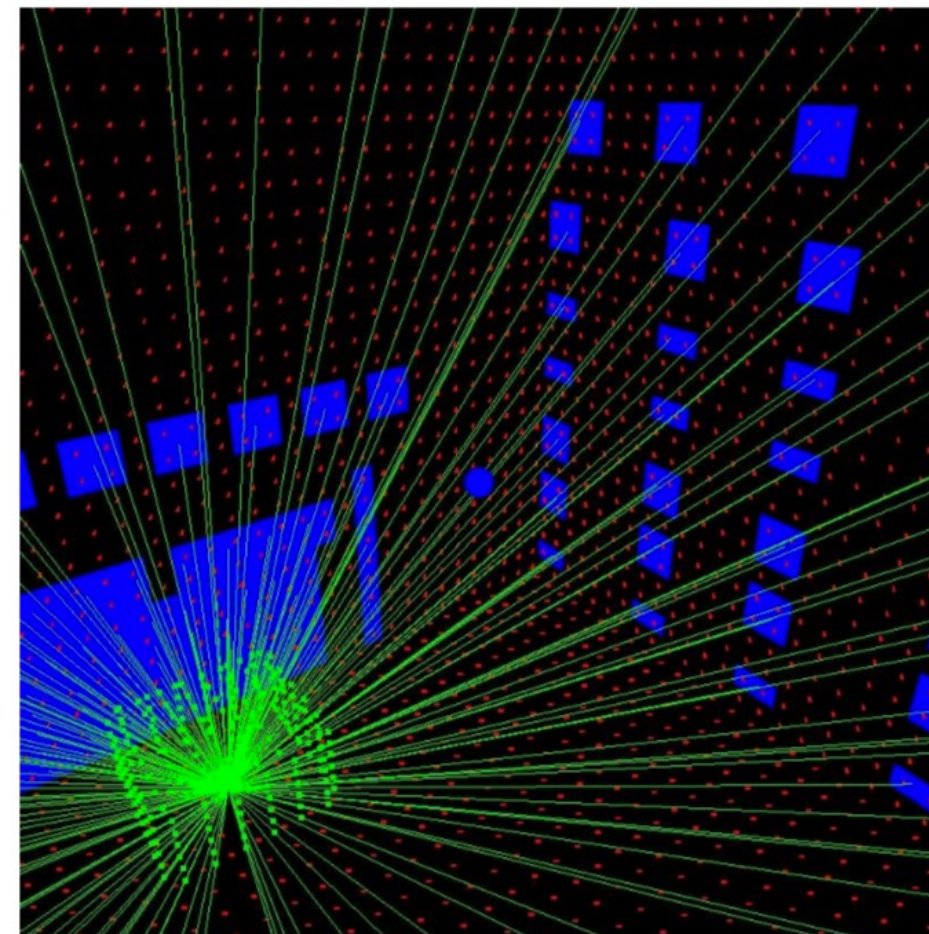
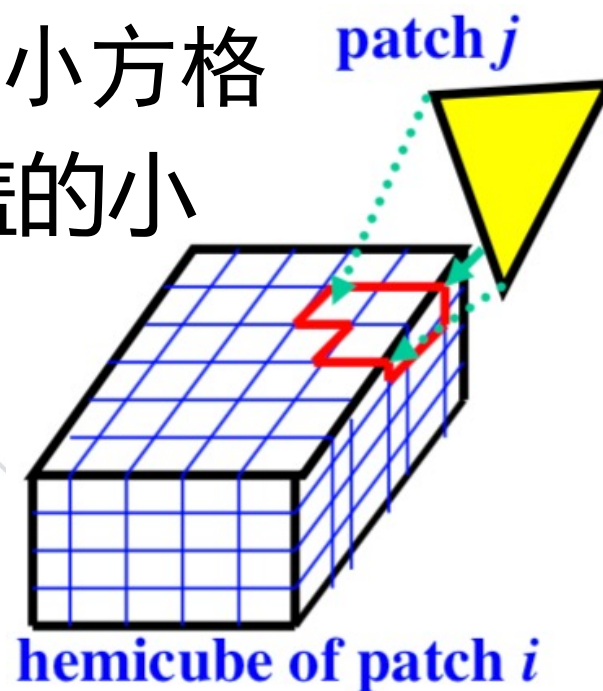
- 等式 $i$ 的residual  $e_i$  可视为小块 $i$ 上尚未发射的能量
- 在每次迭代，我们找到尚未发射能量最多的小块，并将其发出
  - 迭代后，小块 $i$ 上尚未发射能量为0 ( $e_i = 0$ )
- 不断重复此过程，直到没有能量可发射
- Southwell relaxation求解方程组为发射能量的过程





## • Radiosity: 计算form factors

- 假设每个小块都向覆盖在其上的一个半球（hemisphere）均匀发射能量
  - 实现中常用hemicube代替半球（Cohen and Greenberg方法）
- 将小块*i*的hemicube划分成小方格
- $F_{ij}$ 由小块*j*在*i*的投影所覆盖的小方格数量决定



## • Radiosity: 计算form factors

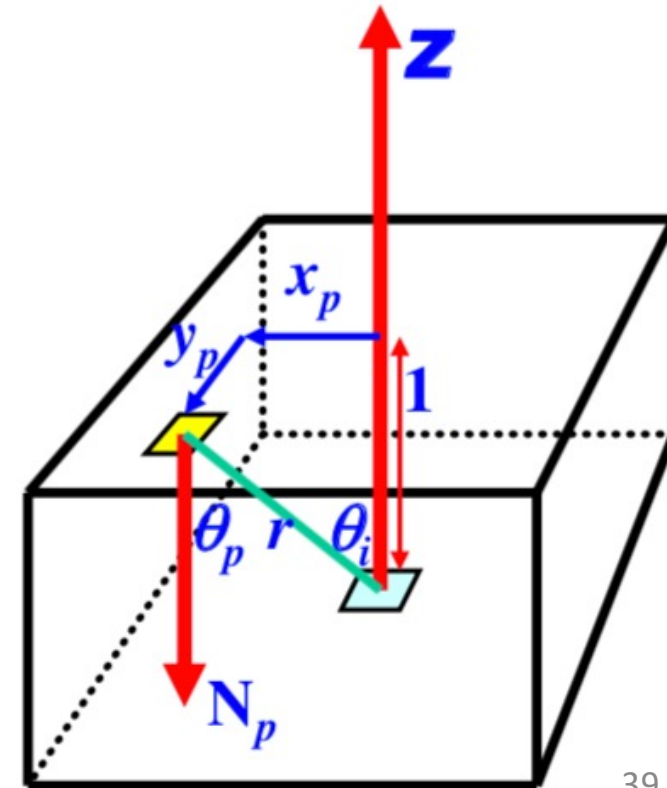
- 实现中，并不计算从*j*到*i*的hemicube的投影，而是从*i*出发向hemicube上小格发出射线，求该射线与其他平面的交点（假设该射线将*i*的能量传输至*j*）
- 由于小格与*i*之间存在夹角，需要计算每个小格对应的form factor
  - 正对*i*的小格接收到来自*i*的能量较多

- 使用余弦计算投影后的强度 $\Delta F_p = \frac{\cos \theta_i \cos \theta_p}{\pi r^2} \Delta A$

- 当*i*的法向量方向沿*z*轴方向时

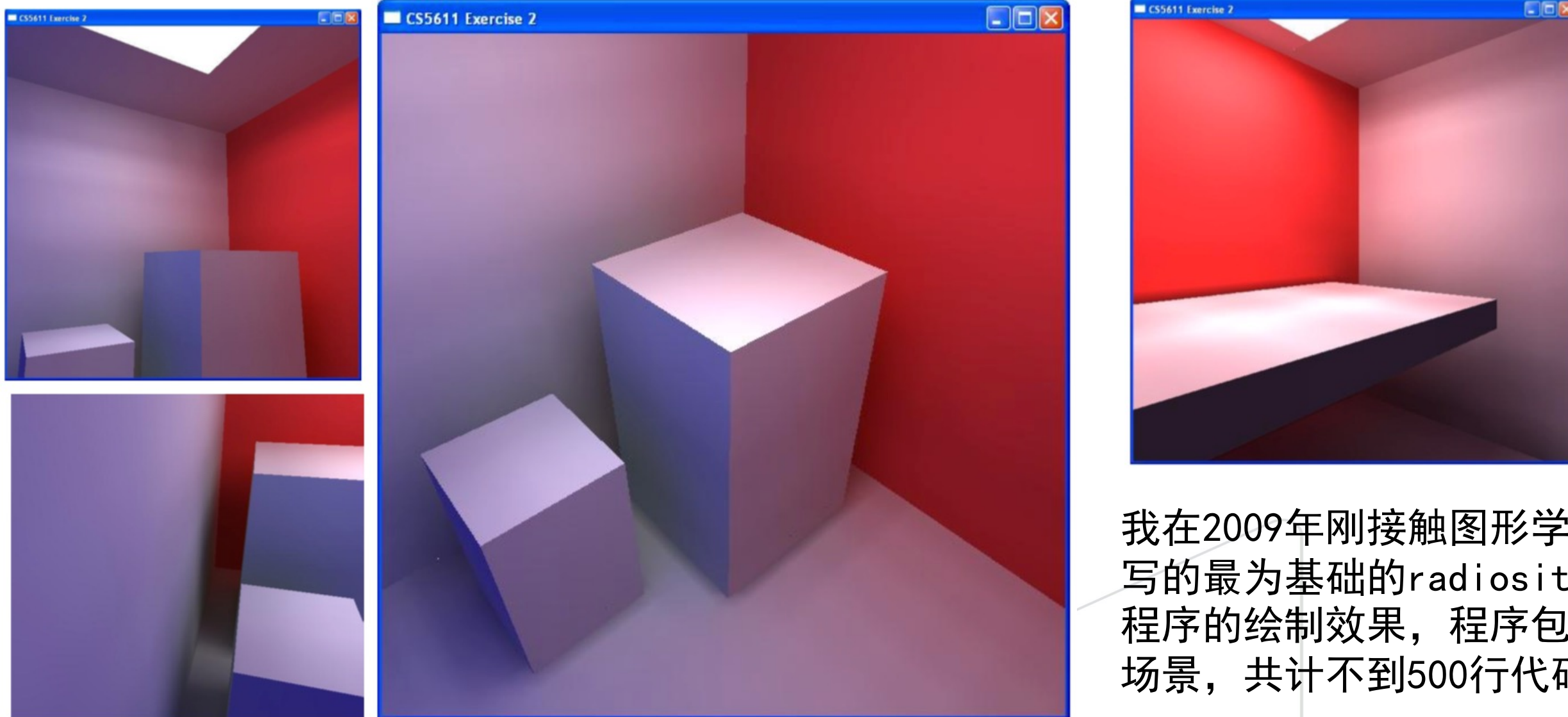
$$r = \sqrt{x_p^2 + y_p^2 + 1}, \cos \theta_i = \cos \theta_p = \frac{1}{r}$$

- 因此， $\Delta F_p = \frac{1}{\pi(x_p^2 + y_p^2 + 1)^2} \Delta A$



- Radiosity: 效果

- Color bleeding, soft light and shadow, indirect illumination





- 局部光照与全局光照简要回顾
- 全局光照
  - Ray tracing
  - Radiosity
- 射线与物体求交
  - 射线与平面，三角形，及球面求交
  - 加速算法



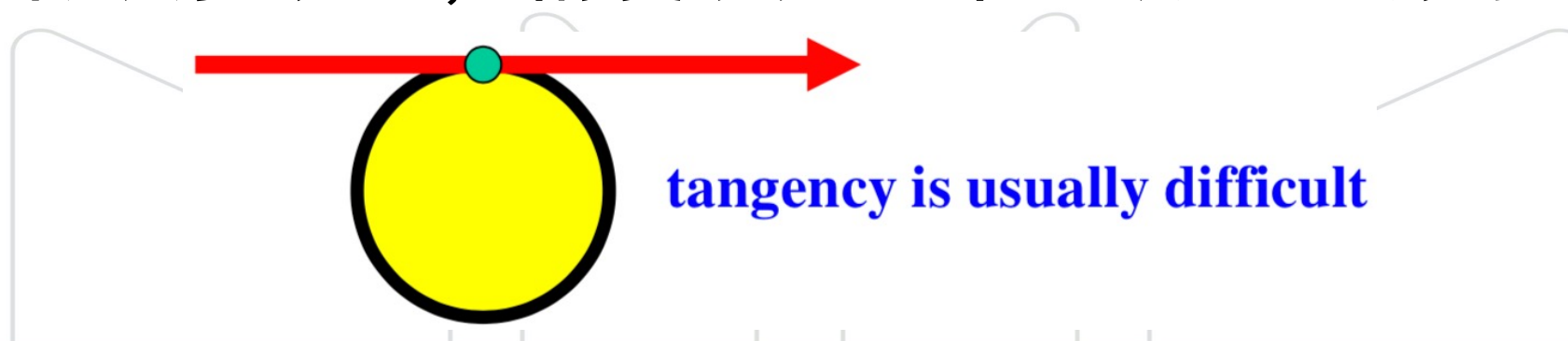
## ◉ 射线与物体相交的通用情况

- 设射线为  $P + tD$ ，其中  $P$  为射线的出发点， $D$  为射线方向
- 设物体对应的曲面为  $f(x, y, z) = 0$
- 需要求的点为射线与曲面相交的点中，离  $P$  最近的点
  - 令  $t^*$  为  $f(P_x + tD_x, P_y + tD_y, P_z + tD_z) = 0$  所有  $t \geq 0$  的实数解中最小的一个
  - 则  $P_x + t^*D_x, P_y + t^*D_y, P_z + t^*D_z$  为所求的最近交点
  - 若无  $t^* \geq 0$  的实数解，则射线与曲面  $f$  无交点
- 例， $P = (1, 1, 0)$ ， $D = (2, 1, 1)$ ，曲面为  $x^2 + y^2 + z^2 = 2^2$ 
  - 则需求解  $(1 + 2t)^2 + (1 + t)^2 + t^2 = 3t^2 + 3t - 1 = 0$
  - 解得  $t = \frac{-3 \pm \sqrt{21}}{6}$ ，其中唯一的正值解为  $t = \frac{-3 + \sqrt{21}}{6}$
  - 交点为  $(1 + 2t, 1 + t, t) = (\frac{\sqrt{21}}{3}, \frac{3 + \sqrt{21}}{6}, \frac{-3 + \sqrt{21}}{6})$



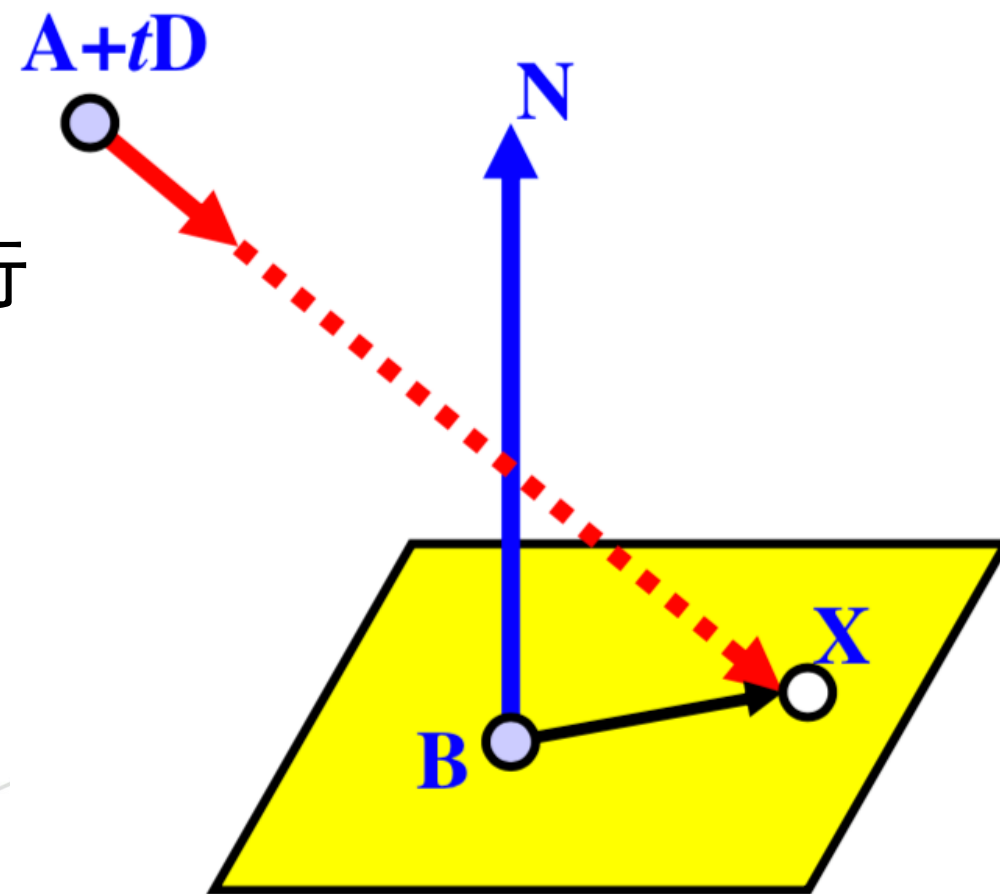
## 射线与物体相交的通用情况

- 然而，Abel与Galois都曾独立证明，高于4次的多项式方程，无法通过四则运算(+, -, \*, ÷)与开方求解
  - 参考Galois theory及Abel-Ruffini theorem
- 因此，当曲面 $f(x, y, z)$ 为通用的高于4次的多项式方程时，我们只能通过近似方法求解（如牛顿方法），但在此情况下，我们很难确定最小的正解
  - 可用Sturm's method确定方程是否在某个区间内有解
- 即便对于低次多项式，精度损失也常造成不正确的解



## 射线与平面相交

- 设射线为  $A + tD$ ，平面上一点为  $B$ ，法向量为  $N$
- 若  $A + tD$  与平面交于点  $X$ ，则有  $X - B \perp N$ ，因此  $[(A + tD) - B] \cdot N = 0$
- 解得  $t = \frac{(A-B) \cdot N}{D \cdot N}$ 
  - 当  $D \cdot N = 0$ （即  $D \perp N$ ）时，射线与平面平行
- 可基于此计算射线与三角形、四边形、网格的交点

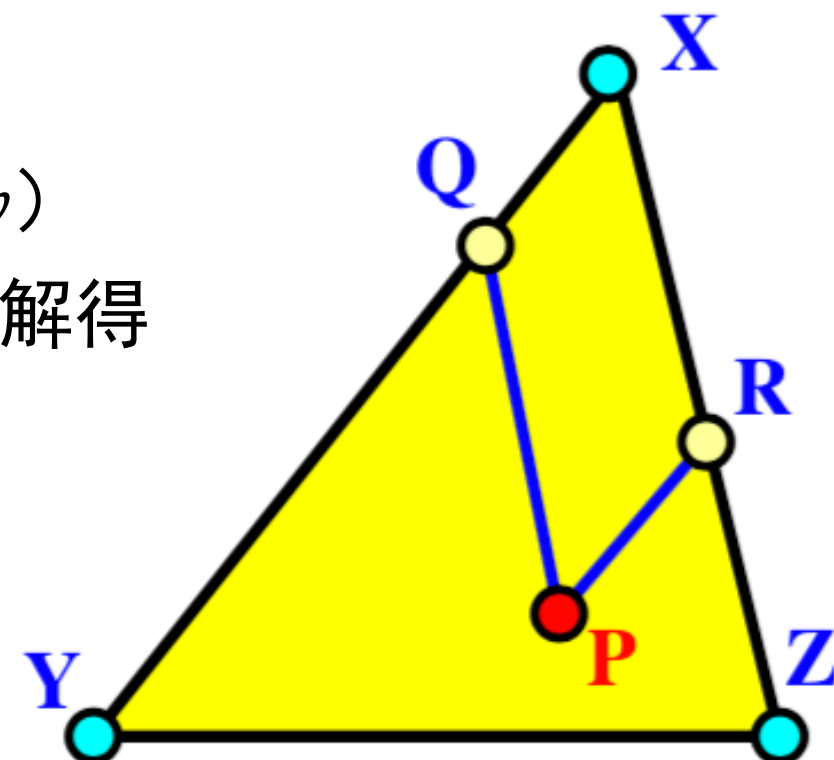
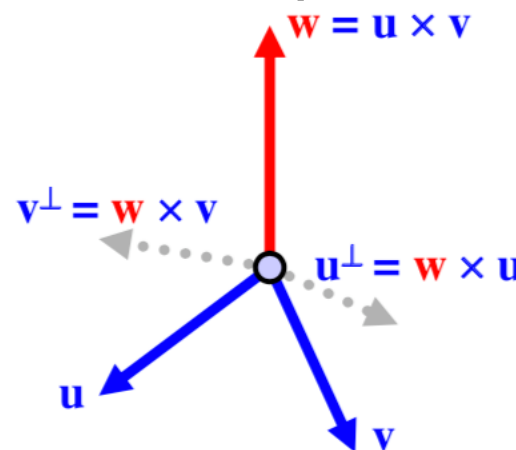


## 射线与三角形的交点

- 在求得射线与平面的交点 $P$ 后，只需判断 $P$ 是否在三角形内部即可
- 三角形内任意一点可写作 $X + \alpha \cdot u + \beta \cdot v$  (barycentric坐标)
  - $X$ 为三角形的一个顶点， $u, v$ 为从 $X$ 出发指向另两个顶点的向量
  - $0 \leq \alpha \leq 1, 0 \leq \beta \leq 1, \alpha + \beta \leq 1$
- 求解barycentric坐标 $(\alpha, \beta)$ 
  - 设 $u^\perp, v^\perp$ 为垂直于 $u, v$ 的两个向量 (即 $u^\perp \perp u, v^\perp \perp v$ )
  - 由 $(P - X) \cdot u^\perp = (\alpha \cdot u + \beta \cdot v) \cdot u^\perp = \beta \cdot (v \cdot u^\perp)$ 可解得

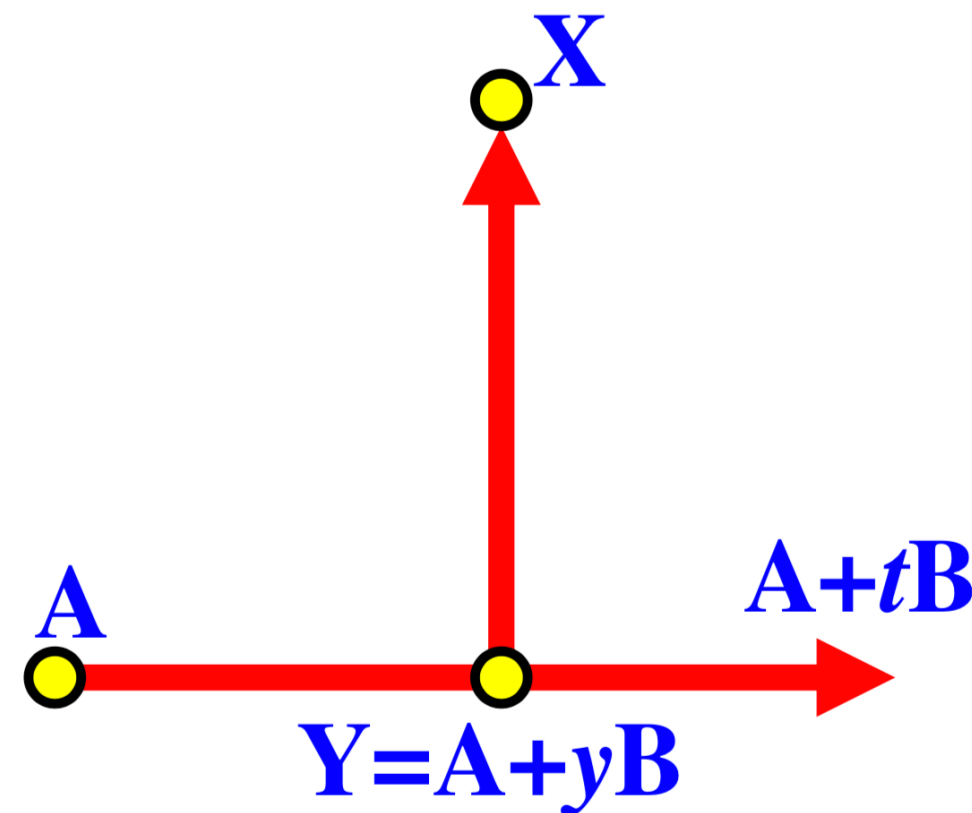
$$-\beta = \frac{(P-X) \cdot u^\perp}{v \cdot u^\perp}$$

- 同理 $\alpha = \frac{(P-X) \cdot v^\perp}{u \cdot v^\perp}$



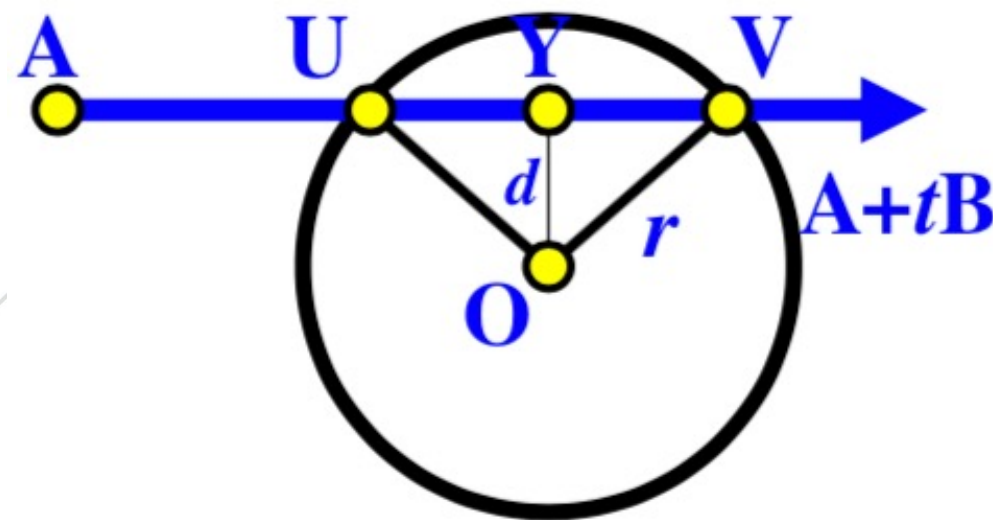
## 点与射线的距离

- 设射线  $A + tB$  上离点  $X$  最近的点为  $Y$ ，则  $XY$  间连线与  $B$  垂直
- 因此有  $(X - Y) \cdot B = 0$  即  $X \cdot B - (A + tB) \cdot B = 0$
- 解得  $t = \frac{(X-A) \cdot B}{|B|^2}$
- 将  $t$  代入可求得  $Y$
- 点到直线的距离即为  $|X - Y|$



## 射线与球面的交点

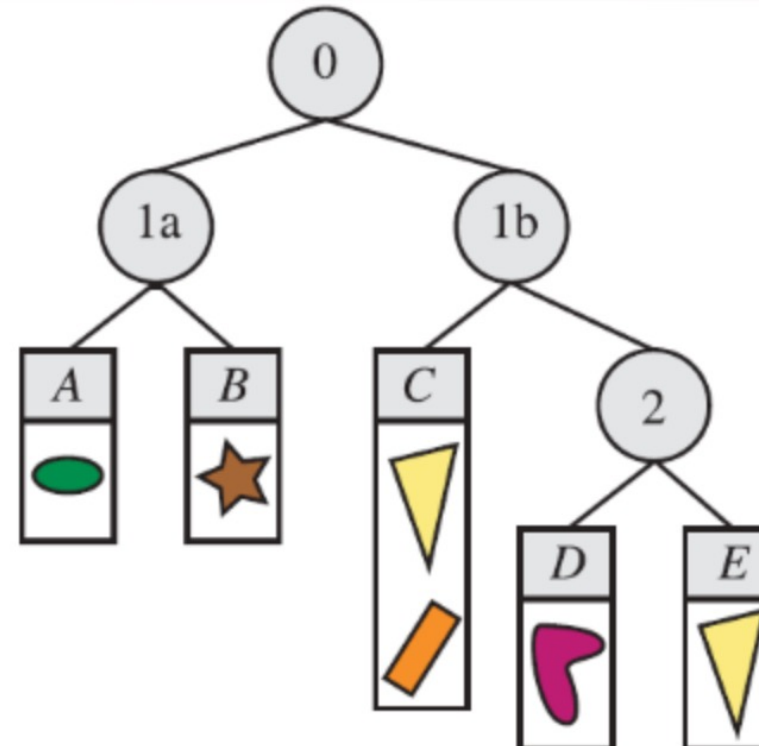
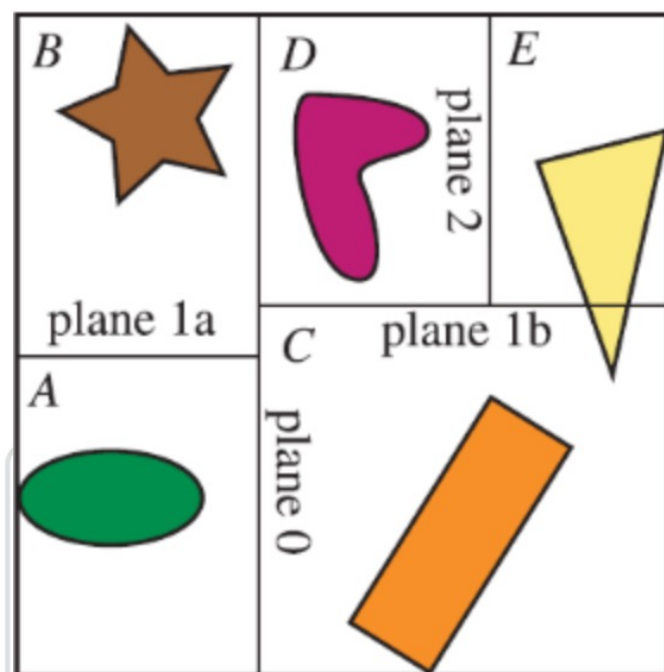
- 设球心为 $O$ ，球面半径为 $r$ ，射线为 $A + tB$
- 计算 $A + tB$ 与 $O$ 的距离 $d$ 
  - 当 $d > r$ 时，射线与球面无交点
  - 当 $d = r$ 时，射线与球面有一个交点（相切）
  - 当 $d < r$ 时，射线与球面有两个交点
- 令点 $Y$ 为 $A + tB$ 与 $O$ 最近的点，则两个交点 $U, V$ 到 $Y$ 的距离为 $k = \sqrt{r^2 - d^2}$ ，即 $U, V$ 为 $Y \pm kB/|B|$





## 减少交点计算

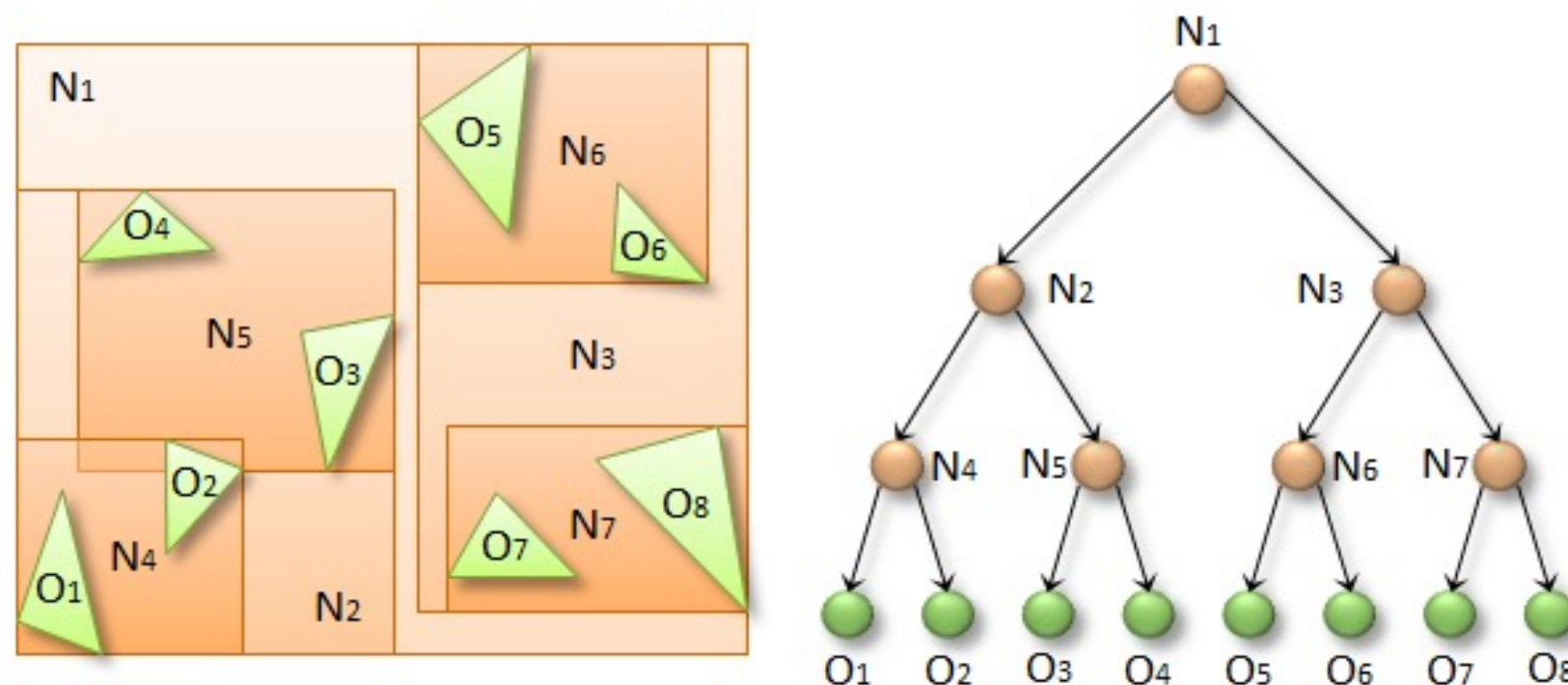
- Bounding volume hierarchies与space subdivision
- Space subdivision举例: kd-tree
  - 八叉树 (octree) 的高维扩展
  - 每次沿一个新的方向将数据切分成两半



## 减少交点计算

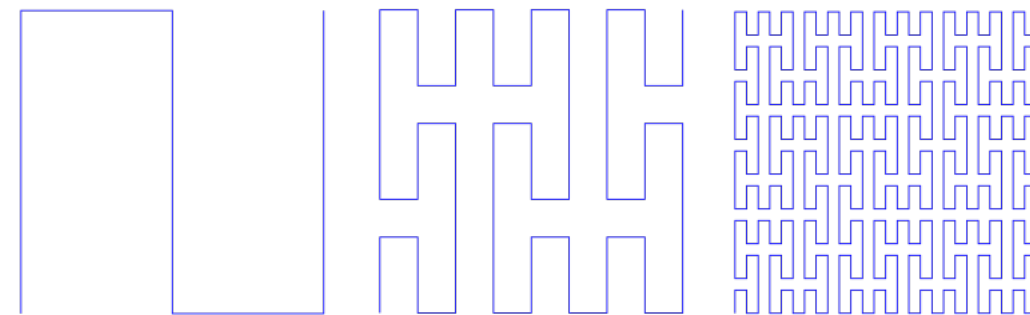
### – bounding volume hierarchy (BVH)-tree

- NVIDIA最新显卡中硬件加速ray tracing即基于此算法
- 使用包围盒（bounding volume）将物体分配到树的节点中
- 在查找最近邻时，先判断与包围盒的距离

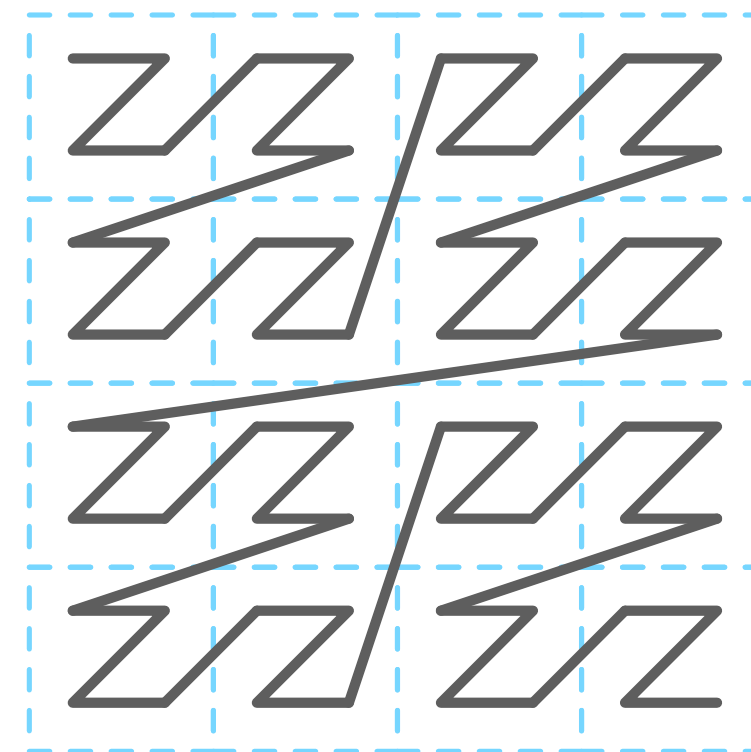
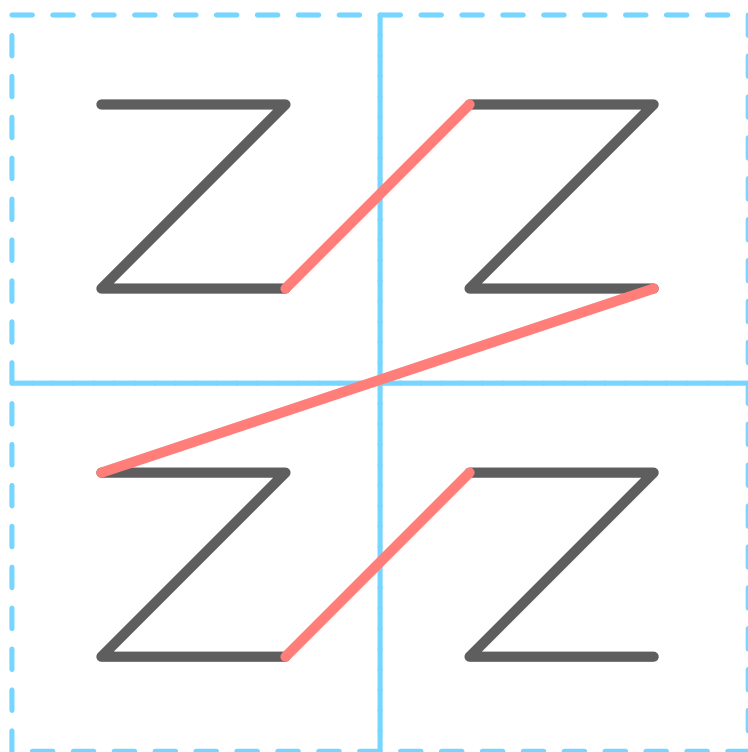
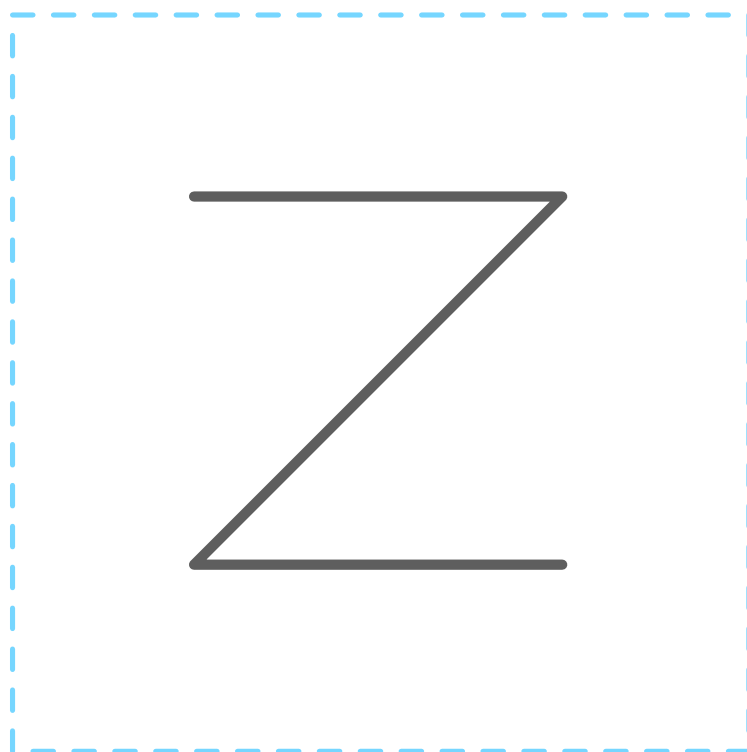


## 构建BVH-tree (Karra's algorithm)

- 1. 将所有几何体沿空间填充曲线排列
  - 沿 Z-order curve 排列
  - 使用一维曲线填充高维空间
  - 源于分形几何



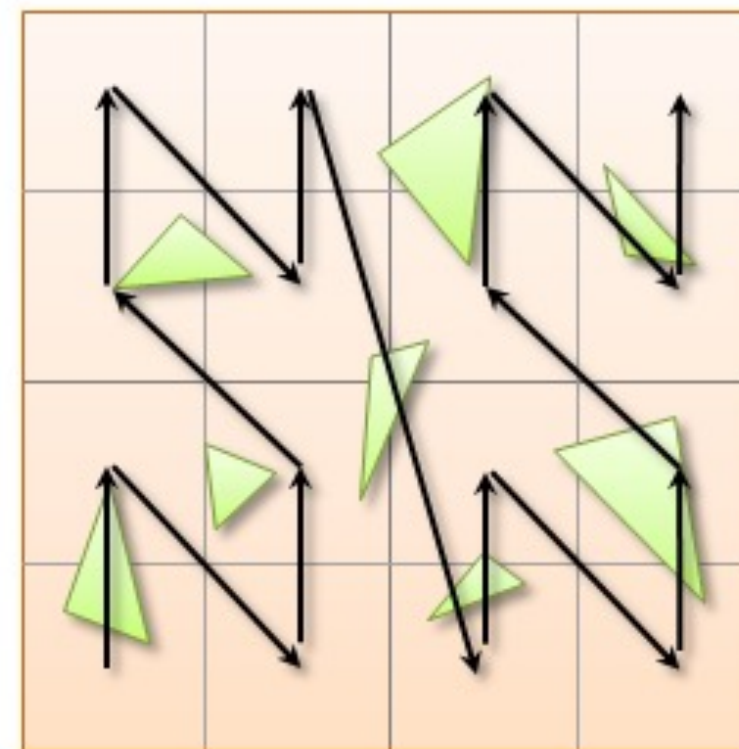
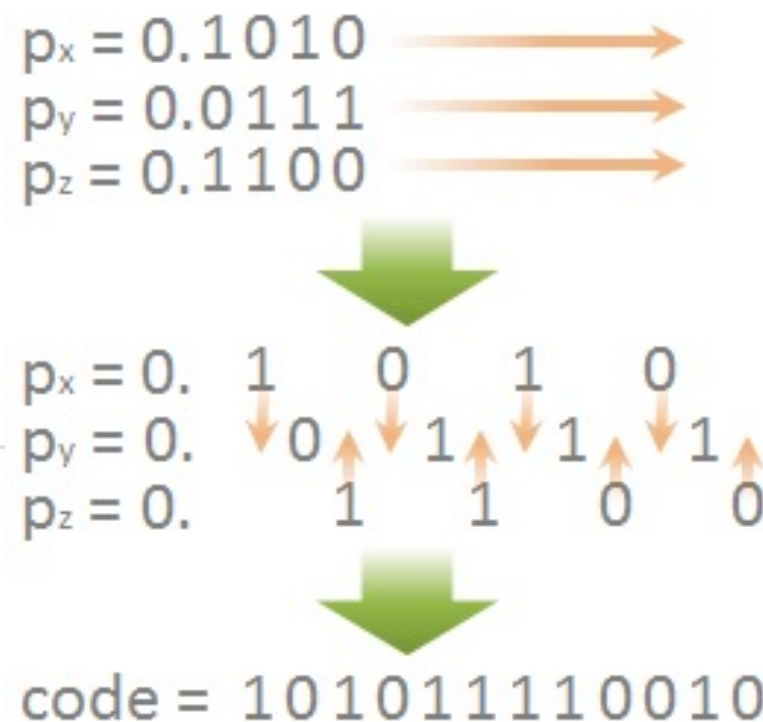
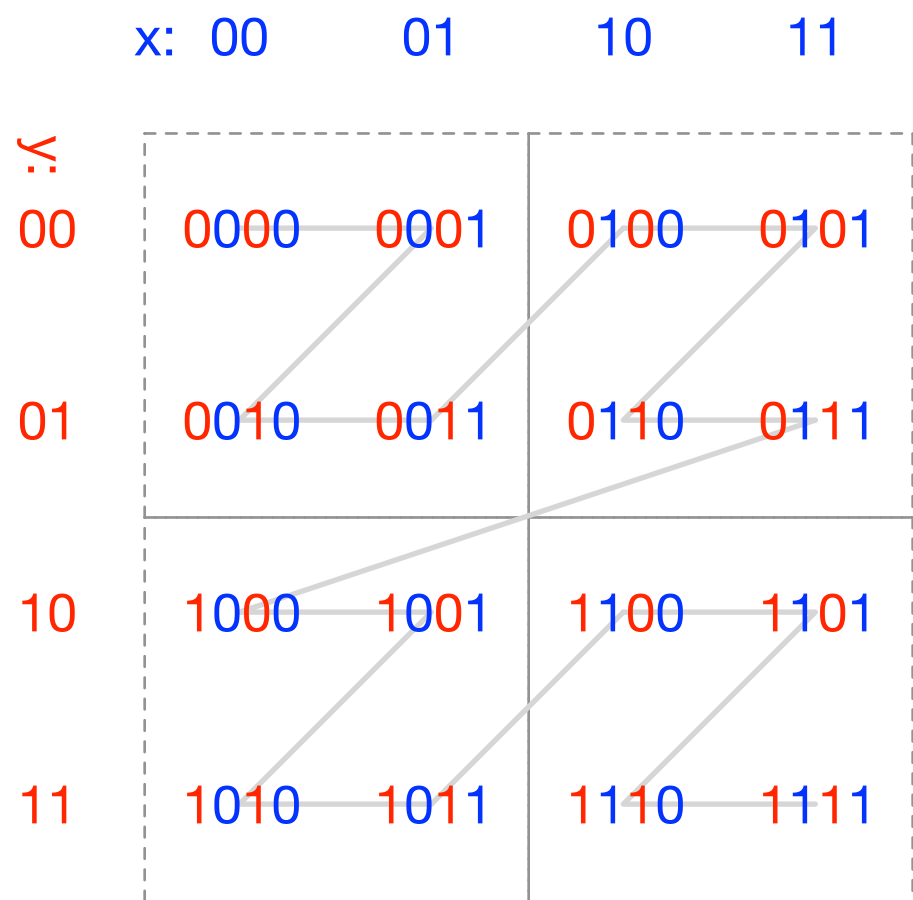
Peano Curve. 图片来自Wikipedia.



## 构建BVH-tree (Karra's algorithm)

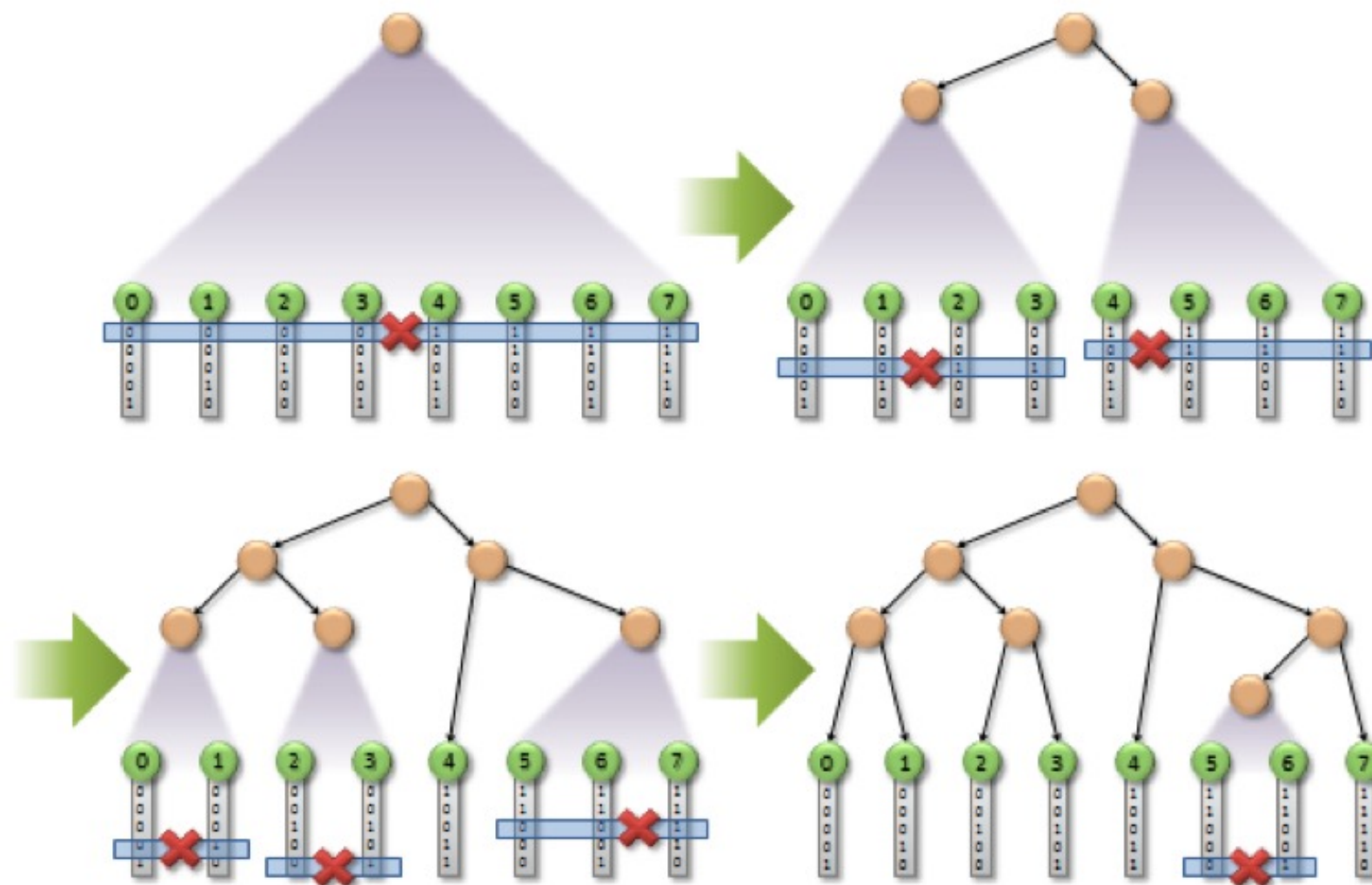
### – 1. 将所有几何体沿 Z-order curve 排列

- 将三维坐标转为Morton codes
- 将几何体按其Morton codes排序 (radix sort)





- 构建BVH-tree (Karra's algorithm)
  - 2. 将排列好的几何体分段
    - 根据Morton codes最高不同位自顶向下分段

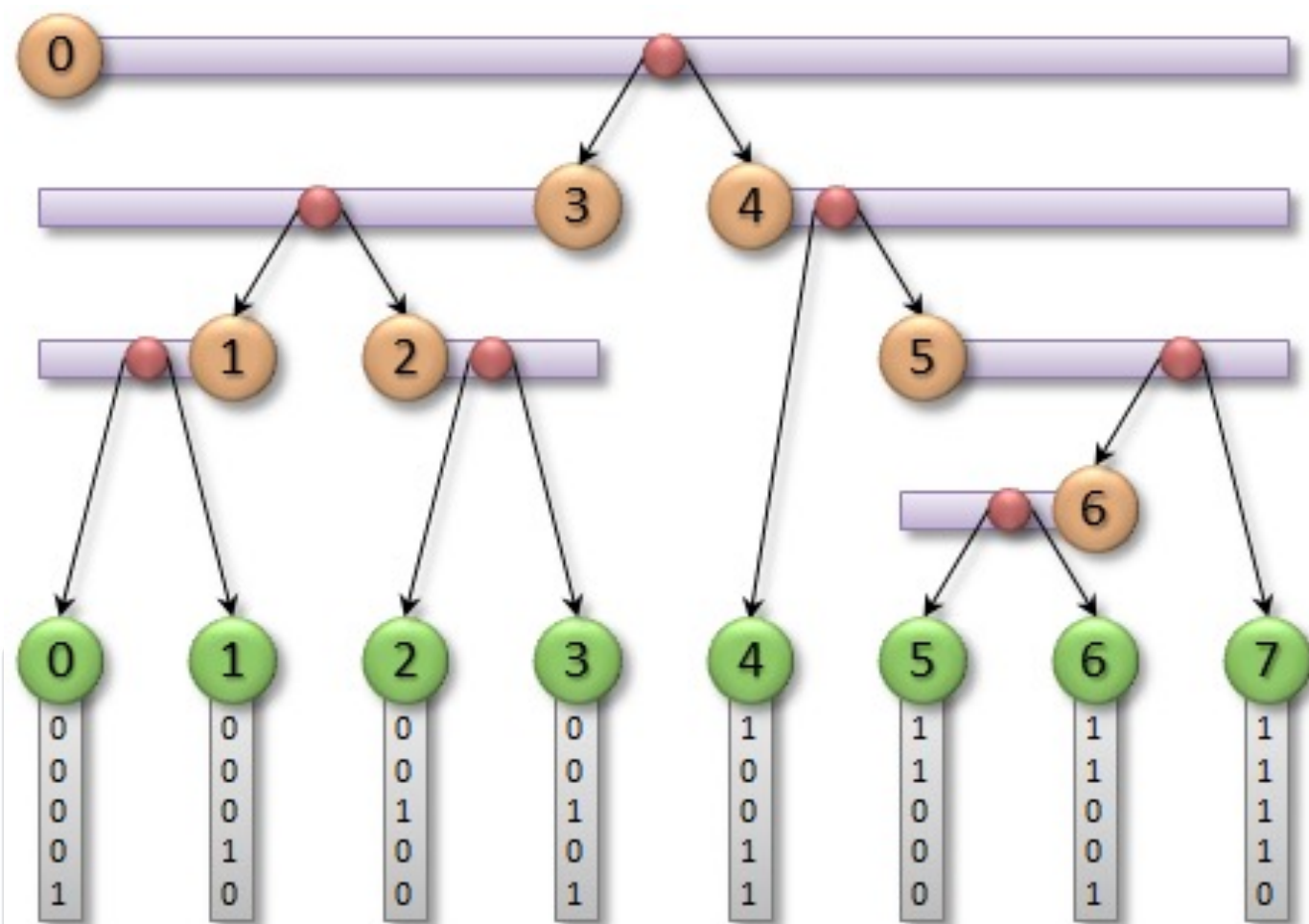


图片来自 <https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-traversal-gpu/>



## 构建BVH-tree (Karra's algorithm)

- 2. 将排列好的几何体分段
  - 提升并行效率



图片来自 <https://devblogs.nvidia.com/thinking-parallel-part-iii-tree-traversal-gpu/>

## ◉ 局部光照下产生阴影

- OpenGL模型本身不产生阴影
- 以光源为视角在需要产生阴影的平面上绘制黑色物体（阴影）
- 无法产生间接照明导致的阴影或复杂形状上的阴影

## ◉ 全局光照

- Ray tracing从视角出发递归追踪光线来源及反射光线的物体，从而计算每一道视线最终捕获的颜色
- Radiosity将物体切分成小块，并将小块间能量传递组织成线性方程组，从而计算每一个小块最终呈现的颜色

## ◉ 线与物体求交

- 活用射线与物体的向量表现形式；使用空间切割算法减少计算

# Questions?

